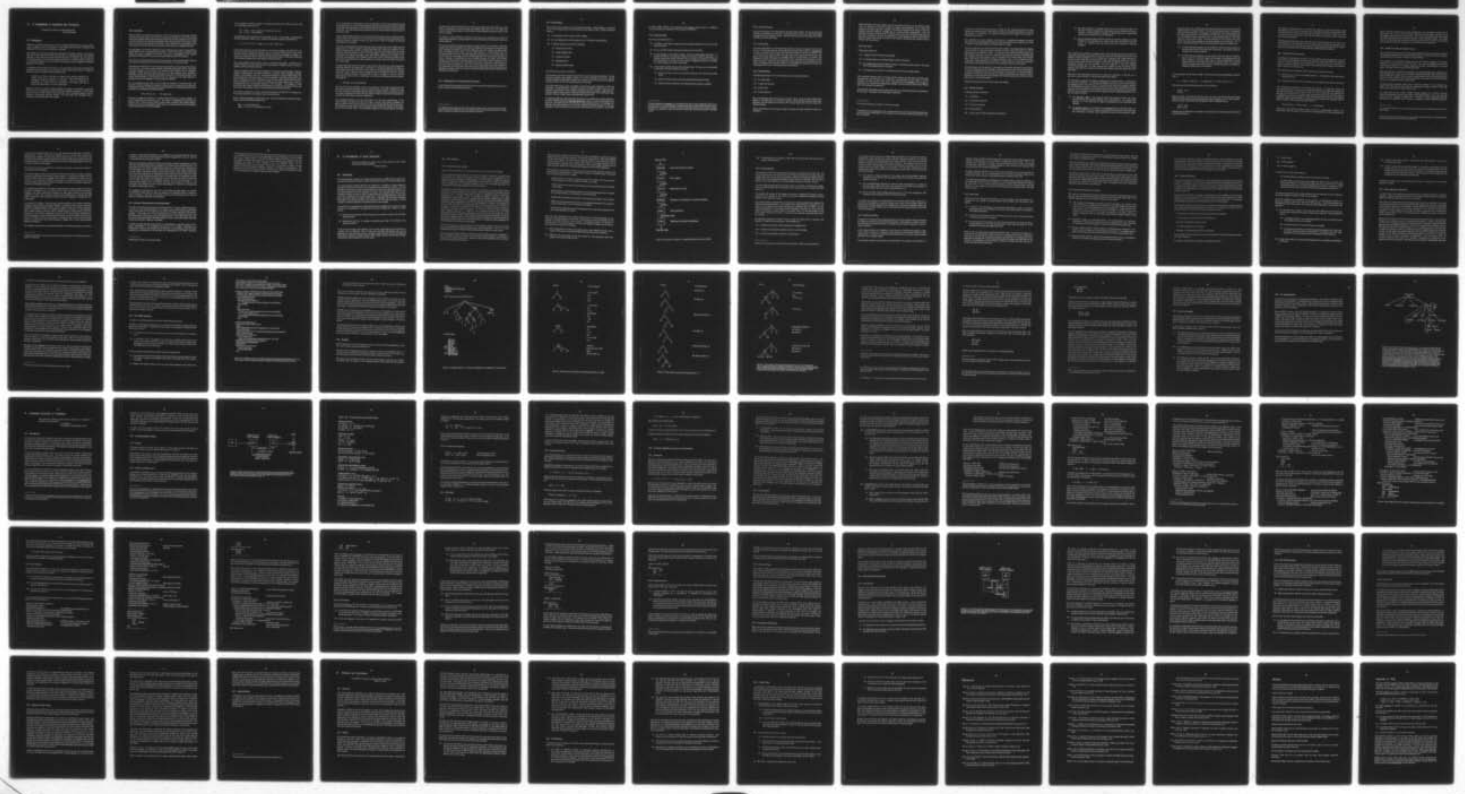
AD-A058 872 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
FORMALIZATION AND AUTOMATIC DERIVATION OF CODE GENERATORS.(U)
APR 78 R G CATTELL F44620-73-C-0074
UNCLASSIFIED CMU-CS-78-115 AFOSR-TR-78-1248 NL

1 OF 2
ADA
058 872



CMU-CS-78-115

LEVEL II

Formalization and Automatic Derivation of Code Generators

R. G. G. Cattell

April 1978

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
AFOSR-TR- 78-1248		

4. TITLE (and Subtitle)

FORMALIZATION AND AUTOMATIC DERIVATION OF CODE GENERATORS

5. TYPE OF REPORT & PERIOD COVERED

Interim rept.

6. PERFORMING ORG. REPORT NUMBER

CMU-CS-78-115

7. AUTHOR(s)

R.G.G. Cattell

8. CONTRACT OR GRANT NUMBER(s)

F44620-73-C-0074

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Carnegie-Mellon University
Department of Computer Science
Pittsburgh, Pennsylvania 15213

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

61101E
A02466/7

11. CONTROLLING OFFICE NAME AND ADDRESS

Defense Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, Virginia 22209

12. REPORT DATE

April 1978

13. NUMBER OF PAGES

130

14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*

Air Force Office of Scientific Research/NM
Bolling AFB, Washington, DC 20332

15. SECURITY CLASS. *(of this report)*

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This work is concerned with automatic derivation of code generators, which translate a parse-tree-like representation of programs into sequences of instructions for a computer defined by a machine description. In pursuing this goal, the following are presented:

(1) a model of machines and a notation for their description

→ next page

20. Abstract

Cont

(2) a model of code generation, and its use in optimizing compilers.

(3) an axiom system of tree equivalences, and an algorithm for derivation of translators based on tree transformations (this is the main work of the thesis).

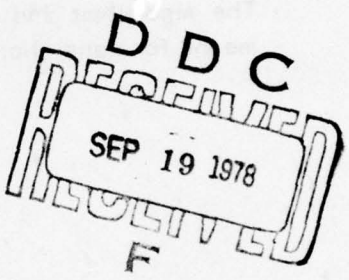
The algorithms and representations are implemented to demonstrate their practicality as a means for generation of code generators.

CMU-CS-78-115

Formalization and Automatic Derivation of Code Generators

R. G. G. Cattell

April 1978



Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract number F44620-73-C-0074 and is monitored by the Air Force Office of Scientific Research.

Abstract

This work is concerned with automatic derivation of code generators, which translate a parse-tree-like representation of programs into sequences of instructions for a computer defined by a machine description. In pursuing this goal, the following are presented:

(1) a model of machines and a notation for their description

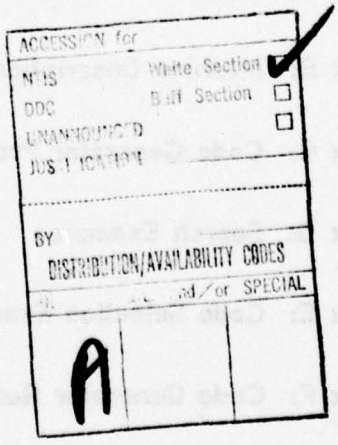
(2) a model of code generation, and its use in optimizing compilers.

(3) an axiom system of tree equivalences, and an algorithm for derivation of translators based on tree transformations (this is the main work of the thesis)

The algorithms and representations are implemented to demonstrate their practicality as a means for generation of code generators.

Acknowledgements

I'd like to thank Bill Wulf, Mario Barbacci, and Allen Newell, who served as my advisors at various times during my stay at CMU. They also served on my thesis commitee, along with Alice Parker and Joe Newcomer; I am grateful for all of their comments. Special thanks go to John Oakley and Steve Saunders, for many interesting discussions on this thesis and on other topics. I'd also like to thank Hans Berliner, Bill Brantley, Steve Crocker, Kitty Fisher, John Gaschnig, Steve Hobbs, Dave Jefferson, Bruce Leverett, Bruce Schatz, and Nori Suzuki, who contributed to this work in various ways. The CMU Computer Science Department in general provides a dynamic, friendly atmosphere conducive to research. I'd like to thank the SIGLUNCH group in particular in this regard. Also conducive to good research are the excellent computer facilities, making programming and document production more enjoyable; Brian Reid, Ken Greer, and Craig Everhart maintain software used in the production of this document. Finally, I'd like to thank my wife, Nancy, who has been very supportive, and my cat, Shrdlu, who tends to sit in the middle of whatever I'm trying to do.



1. Introduction

"I have made this letter longer than usual because I lack the time to make it shorter"

- Pascal

1.1. Motivation and Goals

In the past decade there has been increasing interest in reducing the effort to construct compilers. Most of the reasons for increasing interest in compiler generation are fairly obvious, and have been discussed at length by others (Simoneaux[1975], Newcomer[1975]). As new machines and languages are developed, the problem only becomes more acute. In particular:

(1) It is now possible to generate new machine architectures quite easily, both through LSI technology and microprogramming.

(2) High-quality machine code is often desired. This becomes increasingly important as various technological limits are reached.

(3) Relatively large and complex languages are being developed, requiring larger and better compilers.

Progress has been made in the design of compiler-compilers and translator writing systems, particularly with respect to automating the parsing of programming language text into internal notations. Much less progress has been made in automating the second part of the compilation process: translating the internal representation into instructions for the target machine. I believe this failure is primarily due to inadequate formalization of machines and the code generation process, rather than fundamental difficulties in automating the process.

The goal of this thesis is to study and formalize machines and code generators, and using these formalizations, to automatically derive code generators. The latter problem is a special case of a more general problem of automatic programming: given a program and a set of primitives available, how can the program be decomposed into the primitives? In general, the thesis views the problem from this more abstract point of view; later, the specific properties of machine code generation are exploited to make the problem feasible for practical results.

Most of the previous work in the area has been notably unsuccessful in the sense of practical applicability: it is important to consider the performance of the algorithms (speed and optimality), the generality of the machine model, and the relationship between compiler components, compiler generator, machine description, language description, etc. In order to demonstrate the feasibility, performance, and generality of the theories proposed here, working prototypes have been implemented to form as complete a system as possible within the time constraints of the work.

1.2. Background

This thesis necessarily involves relatively disparate areas of computer science: computer architecture, compilers, and automatic programming. It grew out of two projects currently active at Carnegie-Mellon University: the Symbolic Manipulation of Machine Descriptions (SMCD) and Production-Quality Compiler-Compiler (PQCC) groups.

The SMCD project (Barbacci & Siewiorek[1974]) is centered on a data base of machine descriptions that can be shared by many applications thereof. The goal is that the same computer descriptions be used for emulation of machines, automatic generation of assemblers (Wick[1975]), diagnostics (Oakley[1976]), compilers, actual hardware specifications (Barbacci & Siewiorek[1975]), and other applications.

The PQCC group is interested in simplifying and/or automating the construction of a high-quality compiler generating optimized code. The work is concentrating on the machine-dependent aspects of optimizing compilers, a difficult problem which has received little attention. PQCC is using the multiple-phase structure of the Bliss-11 compiler (Wulf et al[1975]) as a starting point for the research.

Some work has been done in the area of code generation in general (Wilcox[1971], Weingart[1973], Simoneaux[1975]). There have been two classes of approach to simplifying production of code generators. The first is the development of specialized languages for building code generators, with built-in machinery for dealing with the common details; this might be referred to as the *procedural* language approach. Early work in this area was done in compiler writing systems (Feldman[1966], Feldman & Gries[1968], McKeeman et al[1970], White[1973]). Also, Elson & Rake[1970] and Young[1974] have concentrated specifically on code generator specification languages and have been relatively successful. The other extreme is the *descriptive* language approach: automatically building a code generator from a purely structural and behavioral machine description. Miller[1971], Donegan[1973], Weingart[1973], Snyder[1975], Newcomer[1975], and Samet[1975] fit this category, to varying degrees. A survey of the above work, particularly as it relates to the goal of automating the production of code generators, can be found in Cattell[1977].

The more important predecessors, including those concerned with code generator generation, tree equivalence, and machine descriptions, will be discussed in the chapters related to their work. Therefore, no more detailed discussion of background will be necessary here.

1.3. Overview

Figure 1 gives an overview of the problem as viewed by this work. Three algorithms and representations are involved:

(1) The formal description of the machine, labelled MOP in the figure, and its extraction from a procedural machine description language such as ISP (Bell & Newell [1971]).

(2) The tabular representation of the parse-tree to machine-code translation, labelled LOP in the figure, and the code generator which uses it.

(3) The algorithms which derive (2) from (1).

These three problems are discussed in the next three chapters, respectively.

In Chapter 2 we formally define *instruction set processors*, the class of machines with which we will deal. A machine description specifies the types and accessibility properties of data on the machine, and the properties of the machine instructions including space/time costs, the format of the binary instruction words, and *input/output assertions* on the processor state. Machine description languages are also discussed.

The input/output assertions, which define the actions of the instructions, are the main component of the machine description used in the remainder of the thesis. A simple assertion, as an example, would be that for an ADD instruction, specifying that it leaves the sum of a register and a memory location in the register.

In Chapter 3, a formalization of the code generation process is proposed. Its relationships with other components of a proposed compiler are explained. A scheme for separating the compilation process into machine-independent and language-independent phases is used, introducing an intermediate canonical parse-tree notation, TCOL (Tree-base COmmon Language, discussed in section 3.2.1). The front end of the compiler translates a language into TCOL. The back end, which is the concern of this thesis, translates TCOL into machine code. As we will see in Chapter 2, the machine instruction actions (i.e., the input/output assertions) are also given in terms of TCOL operators. The proposed code generator is based on a table-driven tree transformation scheme, in which TCOL trees, in a series of steps, are transformed into code sequences on the target machine.

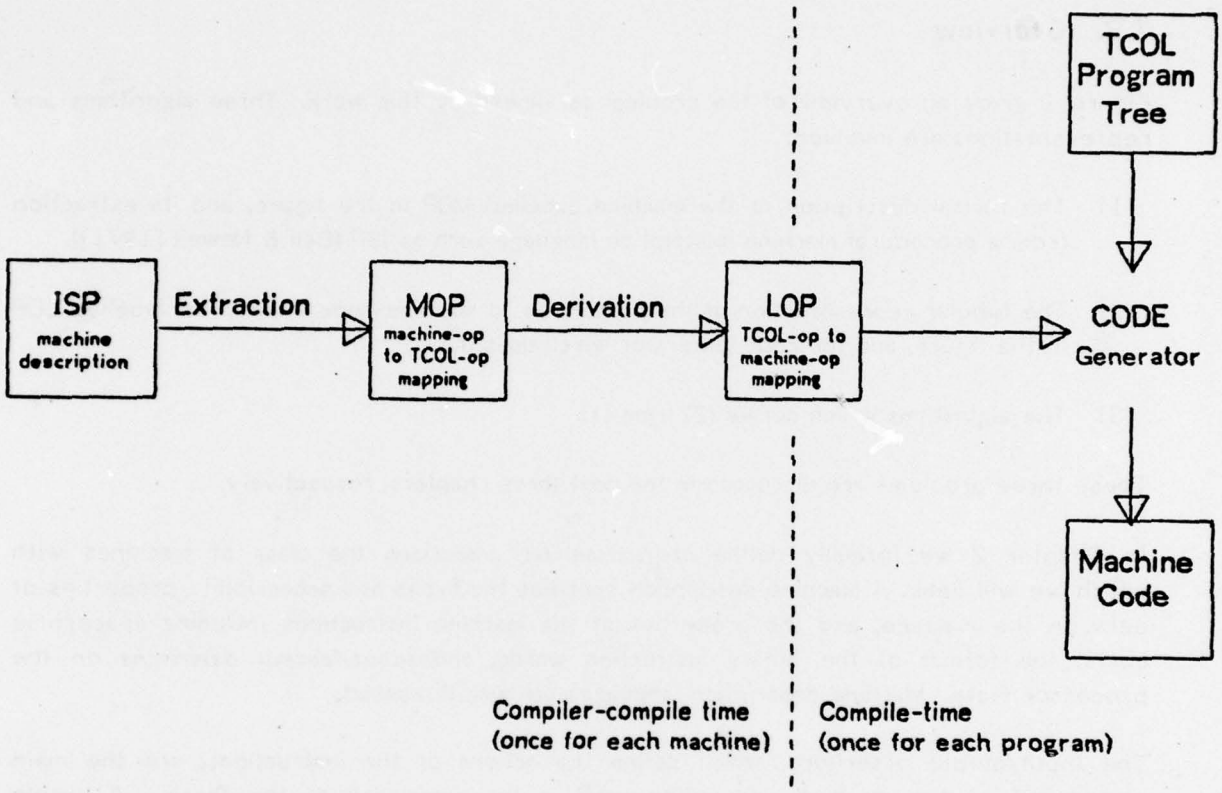


Figure 1: Relationship of the programs and representations proposed by the thesis. In the horizontal direction, the construction of a code generator from the machine description is shown. The MOP is the formal representation of the machine, which could be extracted from a machine description such as ISP. The LOP is the formal representation of the code generation process; the code ge nerator could be table-driven or constructed from the LOP. The code generator derivation process constructs the LOP from the MOP. In the vertical direction, the use of the derived code generator is shown, translating program trees into code for the target machine.

The transformations for this table-driven scheme are given by a set of *templates* in the LOP. A template specifies a TCOL *tree pattern* (a TCOL operator or operators with arguments in certain locations) and *code sequences* to generate (or other actions to perform) when the pattern occurs in the source program tree.

Chapter 4, building on the basis provided by Chapters 2 and 3, proposes algorithms to produce the templates for the code generator from the machine description. The task of this code generator generator is thus to determine the tree patterns defining the special cases recognized by the table-driven code generator, and for each of these patterns, determine the optimal code sequence. A system of axioms is presented to formalize the semantic equivalence of trees (programs). A search algorithm based on transformation of trees into semantically equivalent trees is proposed to derive code sequences. This is a classical heuristic search problem, with givens, a goal, and rules (the axioms) that specify the equivalent problems in the search space. Established methods of artificial intelligence, such as means-ends analysis, are found to perform well on the problem.

Chapter 5 summarizes the research, evaluates the results, and suggest future research.

If the reader is primarily interested in only one of the main three chapters (2, 3, and 4) of the thesis, any of these may be read in isolation. The forward and backward references to related sections should facilitate this. A glossary of terms is provided in the back.

2. A Formalization of Instruction Set Processors

"A problem well stated is a problem half solved"
- Charles Kettering

2.1. Background

Before we can derive code generators or other machine-dependent software, we must define what a machine *is* for our purposes. In this chapter the necessary components and properties of an instruction set processor are defined.

Fortunately, some work has already been done toward formalizing machines. In particular, Bell and Newell's ISP [1971] model has been taken as a starting point for this research. However, a more precise and structured model than theirs is necessary here, because the machine description is to be used for rigorous definition to a computer program rather than for expository purposes. Also, the model we will develop is more amenable for automatic generation of software.

The machine we are dealing with consists of a set of locations, whose values are collectively termed the *processor state*, and an *interpreter*, which executes *instructions* to change the processor state. As defined by Bell and Newell [p 22]:

> "...Some set of bits is read from the program in Mp [primary memory] to a memory within P [the processor] ... This set of bits then determines the immediately following sequence of operations. ...After this sequence of operations has occured, the next instruction to be fetched from Mp is determined and obtained. The cycle repeats itself."

We walk a fine line in making a rigorous definition of a machine in this chapter. On the one hand, we want to include all the machines commonly classified as computers. On the other hand, we want a formal definition that restricts the class of machines enough to make it feasible to automatically generate software. Any useful model must therefore strike a compromise between generality and feasibility.

2.2. Overview

We will define an instruction set processor in terms of seven major components: *machine operations* and *data types*, which specify the operations available on the machine; *storage bases*, *access modes*, and *operand classes*, which specify the locations available on the machine and how they may be accessed as operands; and *instruction fields and formats*, which specify the binary represention of instructions. These components will be defined formally in sections 2.3.1-2.3.7; in this section we present a more informal but readable discussion.

Corresponding to the abstract definitions of the components are syntactic representations: sample MOPs for the a simplified PDP-8 and PDP-11, which will be used as examples in the thesis, are given in appendix B. However, the syntax is independent of the abstract definitions, so the details of this particular representation are left to the appendix.

The model is centered around the instructions themselves. We will formally define these as *Machine-Operations* (M-ops), but continue to use "instruction" in its informal sense.

Associated with each M-op are a set of *input/output assertions*, which express the action of the instruction. An *output assertion* specifies a processor state location and its value after instruction execution as a function of the processor state locations prior to instruction execution. Paired with each output assertion is an *input assertion* which specifies a conditional function of the processor state locations (also prior to instruction execution). The output assertion holds, i.e., the location has the new value, only if this input assertion is satisfied (its value is unchanged if no input assertion is satisfied).

For the purposes of this work, we must relate the assertions to program trees. The assertions will be *represented* as program tree patterns which correspond to the action the instruction performs. In these terms, the previous definition of an input/output assertion is equivalent to a conditional assignment statement:

$$\text{If } B_i \text{ then } L_i \leftarrow E_i \qquad \text{for assertion } i$$

where B_i is a boolean expression, L_i is a location expression, and E_i is a value expression. B_i may be identically true, giving just $L_i \leftarrow E_i$. The utility of representing the assertions as program trees is that we will be able to "match" these as pattern trees against program trees which an M-op could implement. Both program trees and assertion trees are expressed in TCOL (section 3.2.1) to make this possible.

As an example of an M-op, consider an "Increment and Skip if Zero" (ISZ) instruction, with two input/output assertion pairs:

(1) Z←Z+1 (input assertion identically true)
(2) If Z=-1 then PC←PC+1

As mentioned, these assertions are represented as trees. In this thesis, a parenthesized LISP-like notation is used for trees. The previous assertions would be represented as:

```
(; (← Z (+ Z 1)) (=> (EQL Z -1) (← PC (+ PC 1))))
```

Instructions need not refer to fixed locations. In the above assertions, the operand "Z" may be any location in the primary memory. As we will discuss shortly, there is even more flexibility on most machines: "Z" might be accessed indirectly or by indexing, for example. To specify the classes of locations that may be instruction operands, a three-level mechanism will be defined: *Storage Bases*, *Access Modes*, and *Operand Classes*.

The actual locations of the processor state are the *Storage Bases* (SBs). The SBs may be simple locations of various sizes, such as an accumulator or condition code, or arrays of locations, such as the primary memory.

There are typically several choices for operands of an instruction, corresponding to different modes of addressing on the machine: indexed by a register, indirect through a memory location, an accumulator, and so on. These will be referred to as *Access Modes* (AMs). The correspondence of the access modes to the M-op operands is specified by *Operand Classes* (OCs). Specifically, each OC corresponds to a set of access modes, and each operand of an M-op corresponds to an OC. Any of the specified set of access modes may be used in fetching/storing the corresponding operand of the instruction. For example, an ADD instruction might require a general-purpose register as the operand receiving the result, and allow either an immediate constant or a memory location as the other operand.

In the earlier example, "Z" is an OC. For the ISZ instruction on the PDP-8, "Z" corresponds to two access modes: direct and indirect access to primary memory.

Access modes, like M-ops, are specified by trees. The actual definition for these two access modes, from the PDP-8 MOP, happens to be

```
%Mp: (<> Mp $1:#8 0 12)
%@Mp: (<> Mp (<> Mp $1:#8 0 12) 0 12)
```

It is not important to understand the syntax at this point: The first line defines the access mode "Mp" to be an access to the storage base "Mp" (primary memory) at a location defined by "$1:#8" (which just means an 8-bit constant) using the full 12 bits of the word starting at position 0. The second line defines an indirect access, in which the index into Mp is given by an access to Mp.

Also associated with an OC are a set of assertions which specify how the different access modes relate to the actual bits of the instruction. On the machine, the OC corresponds to the effective address calculation process; you might think of "OC" as standing for Operand Computation in this context.

The main two advantages of using these three levels (storage bases, access modes, operand classes) are (1) their usefulness in generating (good) code, and (2) the reduction in the effective number of instructions to be dealt with. The latter is described further in section 2.3.5: the OCs allow M-ops to refer to a (limited) class of actions, namely the same action performed with different operands, rather than exactly specifying the locations in the processor state involved. This variability surfaces later in the fact that access modes and M-ops and are *parameterized* to pass information to the OCs which will specify the actual instruction fields.

Parameterized trees prove to be useful in many parts of this thesis, so a short explanation would be helpful to the reader here. A *parameter* is a variable associated with a node in a tree pattern, which, when referred to later, represents the program tree node that matched the pattern node. These are used, for example, to refer to the address (value) for a symbol node that matched a pattern in the generation of code, or to specify in a pattern that the same subtree must occur two or more times. In the case of an ADD instruction, the pattern tree might be

```
(← $1:Reg (+ $1:Reg $2:Opnd))
```

"$1" and "$2" are the parameter names in this expression. $1 is used to indicate that the destination and operand registers must be the same register, and both $1 and $2 will be used for later reference in determining the actual fields (e.g., register number or mode bits) of the instruction (specified by the OC's "Reg" and "Opnd"). Details of the syntactic representation of trees and tree patterns are given in Appendix A.

The preceding overview is all one really needs for a first-order understanding of the machine model from the point of view of this thesis. However, to be rigorous, complete, and general enough to deal with real machines, a much more complex definition will be necessary. For example, we must specify how the abstract M-ops and OCs are encoded in the binary bits of the instruction words. This necessitates dealing with instruction formats (including

variable-length ones), bits determined by the M-op (e.g., opcode), the access mode (e.g., mode bits), or *parameters* of the access mode (e.g., address field); and so on. The reader may choose to ignore these details for the purpose of understanding the important results of the thesis, if desired.

In addition to the input/output assertions, each M-op has an associated *cost* (used for determining optimality), a *mnemonic* (used for human convenience), an *instruction format*, and a *field-value list*.

The *instruction format* specifies the instruction fields[1] used by the instruction. Instruction formats are used in conjunction with the last component of the M-op, the *field-value list*, which specifies the values the instruction fields must have. For example, we might assert that an Opcode field has value 7 for an "ADD" instruction. In the simplest case the instruction format is a list of instruction fields, the field-value list is a list of constants, the two lists are the same length, and the corresponding fields have the absolute constants as values. Because, as mentioned, the M-ops represent classes of actions with different operands, we also require a mechanism to give instruction fields values dependent on the *actual* program tree that is the "instantiation" of this M-op (the particular action desired of the class this M-op represents). That is, for different access modes and actual addresses in the instantiation of the instruction, the fields must have appropriate values; these values are specified by the OCs. How this is done, by matching subtrees against access modes, will be described in section 2.3.7.

2.3. Components of an Instruction Set Processor

In the following seven subsections, the instruction set processor components discussed in the previous section will defined more precisely.

---

[1] The possible instruction fields, and their sizes and positions within instruction words, are separately specifed. Instruction formats are distinguished as part of the machine model (rather than simply referring to a list of fields) because real machines are normally organized around a small set of such formats.

2.3.1 Storage Bases

The *processor state* consists of a set of *Storage Bases* (SBs). A Storage Base is an ordered set of 1 or more *words*, each word consisting of a fixed number of bits. The SB is defined in terms of:

(1) A *word length*, in terms of bits (a positive integer)

(2) An *array length*, also a positive integer; this is 1 if the SB is a single location

(3) A *type*; the type must be one of the following:

  (a) Program Counter (PC)

  (b) Primary Memory (Mp)

  (c) *Temporary* location

  (d) *Reserved* location

  (e) *General-purpose* location

Each SB also has a name, for reference.

The processor state must contain exactly one SB of type PC and Mp, respectively. The Mp must be a memory array (length in words > 1) in which instructions are stored. The PC location, previous to instruction execution, must contain an index into the Mp array; Mp[PC] contains the first word of the instruction that will be executed.

The other three types, (c)-(e), are defined in the machine model specifically to allow the generation of code generators. Temporary locations are defined as those which the code generator may destroy without saving in generating code (e.g., condition codes); general-purpose locations may be used if saved; and reserved locations may not be used (e.g., stack pointer). The use of these types will be described in section 4.3.4.

It should be noted that a *word* here does not necessarily correspond to its common definition. A word is defined to be the smallest addressable unit of Mp, i.e., of the size specified by the Mp *word*. This might be less than the size of the smallest instruction. The problems of different data and instruction unit sizes, with different addressing schemes for Mp, and alignment of data and instructions on various word boundaries are discussed in more detail

by Wick [1975]. Dealing with storage bases that overlap in various ways is a relatively involved problem, and is ignored for the purposes of this thesis.[2]

2.3.2 Instruction Fields

An *instruction field* consists of:

(1) A *position*, a non-negative integer giving the bit position relative to the first bit of the instruction word.

(2) A *size*, a positive integer specifying the number of bits in this field.

(3) A *word specifier*, a non-negative integer used for variable-length instructions. It specifies the word of the instruction (relative to the first) in which this field occurs. In emitting code (section 3.2.4), instruction words are output as required for the given instruction fields. (If no fields in a given word are assigned, the word is not output; thus variable-length instructions are possible).

(4) A type, which specifies the use of the instruction field. The type can be determined by the field's use in the instruction interpreter:

   (a) type O: the field is used as an op-code, to determine the instruction executed (M-op)

   (b) type C: the field is used to control the operand selected (access mode)

   (c) type D: the field is used only in data expressions (as constant or address)

---

[2] Overlays (mappings) can be expressed in the formalism fairly easily, either by using an optional 4th component of an SB which allows it to be defined as equivalent to (overlaying) another SB, or by reducing all SB references to common denominators by using sub-word and multiple-word indexing into SBs. The problem which is left to future research is the allocation of these SBs to variables and temporaries; storage allocation precedes code generation in the PQCC compiler.

2.3.3 Instruction Formats

An *instruction format* is an ordered list of instruction fields and OCs. The use of instruction formats is described in section 2.3.7. An *OC format* is the same as an instruction format except OCs may not appear in the list. Instruction formats are used with M-ops, OC formats with OCs, as we will see shortly.

2.3.4 Access Modes

An *Access Mode* (AM) is an expression which specifies a *location* or *constant* which can be used as an instruction operand. Like input/output assertions, access modes are represented by trees. A *constant* tree is simply a leaf specifying the length in bits of the constant. This is referred to as an "open constant", as it may take on any value of the specified size (the actual value being specified by an immediate field of the instruction). A *location* tree describes an access to a location in the processor state. It specifies a storage base, an index (which could be a constant or expression) into the storage base, and a position and size within the indexed word. Appendix A details the representation of constants and locations as trees (using the "*" and "<>" pseudo-operators, respectively).

2.3.5 Operand Classes

An *Operand Class* (OC) is a list of *OC-productions*. Each production specifies:

(1) An access mode

(2) A space and time cost

(3) An OC format

(4) A field-value list

We will occasionally refer to the *access set* of an OC. This is the set of access modes which appear in its productions (i.e., the ones it "allows"). Recall that OCs represent instruction operands, and the access set is thus the set of access modes which may be used in an operand position.

The OC productions will be defined in section 2.3.7 along with M-op productions, which are analogous.

Operand Classes and Access Modes are not an essential component of the machine model. They are defined for "practical" reasons, to simplify code generation and to reduce the number of M-ops to deal with. Consider a two-operand instruction that refers to two OCs. Suppose each OC could be assigned any of n access modes independently. Then the instruction could be written as n^2 instructions which refer only to access modes (on the assertion tree leaves). On some real machines, n is large enough to make OCs essential.[3] If desired, however, the reader may think of the M-ops as referencing the locations directly, without loss of generality.

2.3.6 Data Types

A *data type* consists of

(1) A *length in bits* of the data type encoding

(2) An *abstract domain*: for example integers, reals, or characters.

(3) An *encoding function* which, given an object in the abstract domain, gives a bit string that is the representation of the object.

(4) A *decoding function* which is the functional inverse of the encoding function

Each arithmetic operator used in an M-op assertion specifies the data type it operates upon; the operators have meaning only through their correspondence to data types. It is also important to note that the "data" type is associated with the operator, not the data or locations as in a conventional programming language. Data bits have no type except through their interpretation by operators.[4]

Also implicitly associated with each data type is the set of axioms that apply to the operators on that type. The axioms will be described in section 4.2.

---

[3] For the binary instructions on the PDP-11, n=12 (or so), for example.

[4] This approach is the conventional point of view in computer architecture; on an architecture taking the opposite point of view, such as the Burroughs 6500, it would be necessary to treat the actions of operators as dependent on type bits of the operands.

This thesis will not deal with data types in detail; the only information about data types needed for code generation is contained in the axioms, so the data type portion of a machine description is not required. For complete description of data types, two extremes of approach may be taken.

One approach is to use a notation at the level of a programming language, specifying procedures for the encoding/decoding functions and the operations on the data type in terms of some primitive data type(s), e.g., unsigned integer bit strings.

The other approach is to describe the data types statically, by parameterizing the possible representations in some way, making it possible to specify a data type simply by a set of parameters and an alphabet of types. It is possible to characterize the common integer, real and character data type in terms of a relatively small number of parameters (Cattell[1976]). For example, fixed point numbers may be represented with different bases and fractional point conventions, and different encoding conventions such as two's complement, one's complement, signed-magnitude, and excess-N. Floating point can then be categorized according to the encodings and positions of the (fixed-point) mantissa and exponent.

The advantage of the first approach to describing data types (the procedural representation) is that it is more general; the advantage of the static approach is that we have some hope of a program automatically dealing with the properties of the data type. The best approach is probably to combine the two, specifying the complete algorithm for, say, machine simulation purposes, but asserting particular properties (that could presumably be verified) for other applications of machine descriptions.

Section 3.2.2 describes the use of data types in the compiler.

2.3.7 Machine Operations

A *Machine-operation* consists of:

(1) A mnemonic

(2) A space and a time cost

(3) An instruction format

(4) A field-value list

(5) A set of zero or more input/output assertion pairs:

(a) Each input assertion is a boolean function tree: a tree whose top node is an operator returning a boolean result, and whose leaves are constants or *location specifiers*. A location specifier is normally an operand class, but an access mode will also be permitted here if it is a fixed locaton.

(b) Each corresponding output assertion consists of a *location specifier* (a location modified by the instruction), and a (new-) value tree: a tree whose top node returns an arithmetic value and whose leaves are constants or *location specifiers*.

Each assertion pair specifies a potential change in the processor state affected when the instruction interpreter processes the M-op. The semantics of the assertions are that if the boolean function specified by (a) is satisfied over processor state values previous to execution, then the location specified by (b) has the value specified by function (c) in terms of the processor state previous to execution. There must be assertions for every location potentially modified by the instruction. The only exception is that the assertion "PC←PC+<instruction size>" has been factored out; it applies to every instruction unless explicitly overridden by an assertion on the PC (this simplifies human and machine use of the assertions).

Note that if the input/output assertion set is empty, the instruction is a "No Op", i.e., it performs no action; however, it has cost and takes space in memory.

The input/output assertions thus specify the effect of the instruction when it is executed by the instruction interpreter. The field-value list, together with the instruction format, specify the conditions under which the instruction is executed by the instruction interpreter, and also the correspondence between instruction operands and binary fields. Specifically, the field-value list (a list of parameters and constants) and the instruction format (a list of instruction fields and operand classes) must be identical in length, and the corresponding elements of the lists are paired as follows:

(1) The instruction fields in the instruction format are asserted to have the values specified by the corresponding elements of the field-value list. For example, the constant "7" in the field-value list might be associated with the instruction field "OpCode".

(2) The operand classes in the instruction format indirectly specify instruction fields. As mentioned earlier, the M-ops represent a family of actions, because (a) the operands may correspond to different access modes, and (b) even for one particular access

mode (e.g., indexed by a register), different actual addresses may be involved (e.g., the register number or memory location). To allow this field specification to be separated from the M-op descriptions, a *parameter* is associated with each location specifier in the input-output assertion. This parameter is paired with a particular operand class; i.e., the parameter and operand class occur in corresponding positions of the field-value list and instruction format. The OC-productions for the operand class then specify the actual fields according to the actual operands of the instruction. This, in turn, is finally accomplished by the corresponding OC field-value list and OC format:

(a) As with the M-ops, constants may be assigned to fields by pairing constants in the field-value list with instruction fields in the format. This would be used, for example, for an address mode field.

(b) As with the M-ops, parameters may also appear in the field-value list. However, they are used for a different purpose: the parameters are associated with the *open constants* in the access mode trees, and the corresponding OC format element must be an instruction field which is asserted to have the constant's value. For example, this would be used to relate the operands to an address or immediate constant field of an instruction.

As an illustration of the above, consider the ISZ instruction whose input/output assertion tree is

```
(;  (← $1:Z (+ $1:Z 1))  (=> (EQL $1:Z -1) (← PC (+ PC 1))) )
```

The field-value list and instruction format for ISZ are the lists:

```
(2 $1)  and
(OP Z).
```

What this means is that the OP instruction field has value 2 for ISZ, and the other fields depend on $1:Z, which may be a direct or indirect memory reference. If it is a direct memory reference ("%Mp" access mode) then the OC field-value list and OC format are

```
($1 0)  and
(ADR I.BIT),
```

meaning that the ADR field is the address of the operand ($1 in the %Mp access mode tree) and the I.BIT field is 0.

We define the *field assertions* of an M-op, with respect to a particular (program tree) instantiation, to be the set of assertions about instruction field values resulting from the M-op field-value lists and OC field-value lists as outlined in this section. That is, the field assertions specify the necessary instruction field values to execute a particular instruction with particular operands.

The details concerning the actual instruction bits as outlined above are not crucial to an understanding of this thesis, but the reader should keep in mind that the assertions are in two levels: for each instruction, fields assertions specifying the opcode bits, etc.; and for each possible operand, field assertions specifying mode bits, address fields, etc.

2.4. Instruction Set Processors

In summary, an *instruction set processor* is a set of locations termed the *processor state*, and an *interpreter* which fetches *instructions* from a *primary memory* and modifies the processor state in a fashion specified by the instructions. The entire effect of an instruction is represented by the change in the processor state, and the entire change in the processor state is specified by the instruction.

More specifically, the interpreter iteratively performs the following two steps:

(1) Retrieves one or more words Mp[PC], Mp[PC+1], ... satisfying the field assertions of exactly one M-op.

(2) Changes the processor state as specified by the input/output assertions for that M-op.

This completes the definition of an instruction set processor. The observant reader may have noticed some of the choices made between generality (of the model) and feasibility (of automatic generation), as mentioned in section 2.1. Some of these are fairly obvious; for example, we are restricted to a uniprocessor executing instructions from a primary memory, as opposed to, say, an array processor. Other restrictions are more subtle; for example, the conditions under which instructions are executed (i.e., the criterion the interpreter uses to select an M-op in step (1) above) must be of the form

$$field_1=value_1,\ field_2=value_2,\ \ldots\ field_n=value_n$$

rather than some arbitrary boolean function of fields or storage bases. Some of the assumptions made are important to making the results presented in this thesis possible, and

others could be changed with only minor consequences. The uniprocessor assumption is of the former kind. In the case of the M-op field assertions, any boolean function that is invertible would theoretically be satisfactory. That is, for a code generator we must be able to satisfy (automatically) the conditions asserted, by setting instruction fields[5]. The chosen form of the field assertions allows this to be done easily, while at the same time remaining flexible enough to deal with essentially all conventional machine architectures.

2.5. Relation to Other Descriptive Levels

The reader may be curious as to how the machine model described in this chapter relates to the assembly language level of programming; the model is presented in terms of the binary representation of instructions and data in the actual machine.

The invention of an assembly language given only the machine language for the "bare" hardware is, of course, a creative non-trivial task. There are issues ranging from what the instructions actually are[6], to choosing appropriate syntactic representations for various fields. This topic is well covered by John Wick [1975].

Wick demonstrates a program which produces an assembly language and an assembler for that language, given a derivative of ISP (Bell & Newell [1971]) with some human input to specify mnemonics and the syntax for certain instruction fields. Wick takes advantage of the fact that current computers and assembly languages are very similar in structure, so that the bulk of an assembler can easily be table-driven.

A simple assembly language format, similar to Wick's but obviously not as extensive, was used in this thesis work to make code easily intelligible to humans. However, at the same time we retain the ability to generate binary code.

It is quite easy to get a simple readable assembly language for a machine using the instruction mnemonics and field types given in the MOP description. This is the way the search program in Chapter 4 outputs code. Furthermore, a more sophisticated assembly language can be supported almost as easily, by supplying an output formatting routine for

---

[5] or processor state locations, in the extreme case of a processor whose instruction set depends on the value of some flag or register

[6] This can be subject to human interpretation, for example whether "add" and "add immediate" are treated as different instructions or the same instruction with different operand modes.

each instruction format defined. This is how the code generator described in Chapter 3 outputs code. Further refinements are possible, such as a simple macro notation for describing the translation from the binary fields to textual output. Or, if desired, the MOP definition itself could be changed to describe the machine at the assembly level rather than in binary; however, it is probably a bad idea to throw away information altogether this way, as the description is not as flexible.

The reader may also be wondering how the model of instruction set processors relates to machine description languages.

It is not the intent, in deriving the model of machines for this thesis, to create a machine description specialized to derivation of code generators. If anything, the opposite is true: existing machine description languages were already too specialized to tasks such as simulation. The purpose of the MOP representation is to describe a machine that fits the mold of a model specific enough to make it reasonable for input to an application requiring an instruction set processor description.

The proposed representation does suggest requirements that would be placed on a machine description language for use in this kind of work. Nearly all current machine description languages are simply programming languages with some special features for describing computers or logic.[7] Any particular syntax has been avoided here by defining the MOP tables. However, a short discussion of machine description languages is included here for completeness.

As mentioned in Chapter 1, there are many potential applications of a machine description language: machine simulators/emulators (Barbacci & Siewiorek [1977]), proving correctness of machine language programs (Crocker [1977]), register-transfer level design automation (Barbacci & Siewiorek [1975]), and automatic generation of software such as assemblers (Wick [1975]), diagnostics (Oakley [1977]), peephole optimizers (Hobbs [1976]), and code generators. If all of these applications were driven off the same machine descriptions, it would be possible to write a machine description and obtain all this support software, and even circuit layouts to construct the machine. Although this goal is still a ways off, initial results in several areas have been promising, to suggest that at least *semi*-automatic operation is feasible.

One candidate representation for machine descriptions in the common data base is a machine-

---

[7] ISP (Bell & Newell [1971]) is somewhat more than this, but as mentioned earlier, it is not sufficiently restrictive or specific for our purposes.

readable variant of ISP, ISPS (Barbacci et al [1978]), which was designed with the intent of providing a well-defined, machine-readable description language that could be used for various applications, as required by SMCD.

ISPS has the necessary facilities to include the information required by applications such as this work: the MOP tables could be automatically constructed from an ISPS description. The ISPS description must specify, through special declarations, certain necessary components of the description, such as the program counter (PC), primary memory (Mp), interpretation process, address calculation processes, and instruction formats. Some additional information can be derived automatically by recognizing accesses to the PC, Mp, and instruction fields. The main step that must be performed to construct the MOP table is to symbolically simulate *each* possible path through the interpretation cycle to determine the possible instructions, and for each instruction, the input/output assertions on the processor state [8]. The input/output assertions consist of condition-value pairs which specify new values for the processor state location for each possible path/sequence of branches and assignments through the interpretation process.

For purposes of this thesis, at the time of this writing, the MOP tables are manually constructed; the MOP (and LOP) tables are designed to be readable/writable by people as well as machines. In fact, they are as short to write as the ISPS description, so the only motive for automatic translation is the SMCD goal.

2.6. Syntactic Representation and Implementation

The MOP is syntactically represented in a parenthesized form in a fairly straightforward way. The syntax for the components defined in this chapter is given in BNF notation in appendix B, along with the MOP for the the PDP-11 and the "Mini-S" (a slightly simplified PDP-8), used as examples. These MOPs were manually generated; they may be automatically derived from ISP descriptions at some future time, as mentioned earlier. Partial MOP tables were also derived for the Motorola 6800, Intel 8080, and DEC-10 (to test the machine model with some diverse architectures). These are not included.

It should be noted that the M-ops and OCs are represented as *productions* rather than a simple enumeration of the components as defined in this chapter. Specifically, the input/output assertion tree becomes the left-hand side of the production, and the other components are specified by the right-hand side (recognizable by the "EMIT" keyword),

---

[8] Oakley[1977] is working on such a symbolic simulation.

which specifies the space/time cost, format, mnemonic, and field-value list. This form is used for the MOP because its syntax is now compatible with the representation of the LOP tables described in the next chapter. In the MOP, the RHSs are EMITs; in the LOP, they may be any code sequence (EMITs, code labels, etc.), or even special commands to the compiler. The main advantage of using the same notation is that the MOP/LOP tables can be read/modified/written in several stages of the process of constructing a compiler, without requiring several intermediate notations. However, the syntactic representation is not important to any of the results presented in the thesis, and a different representation might be more appropriate, for example, to improve human readability.

3. A Formalization of Code Generation

"The most valuable of all talents is that of never using two words when one will do [in code generation]"

- Thomas Jefferson

3.1. Introduction

The code generator proposed here, which will be referred to as CODE, takes a program tree as input, and produces as output a symbolic machine code stream with formatting information.

The code generation scheme is based on *tree productions*, or *templates*. A source program is assumed to have been translated into an intermediate tree representation (TCOL). The code generator traverses the program tree, matching each program tree node against patterns on the *left-hand-side* (LHS) of the templates in the LOP. The *right-hand-sides* (RHSs) of the templates matched may specify code to be generated, special compiler actions to be taken, and further matches to be recursively performed. These template/productions, along with other information about the target machine such as addressing and instruction formatting (described in the previous chapter), are specified by the LOP table (see Figure 1), which provides the machine-dependent information required by the compiler.

A description of a code generator without definition of the remainder of the compiler, at least in terms of its relationship to code generation, would be of limited utility. Ideally, we would like to:

(1) Show how the algorithms and data structures can be made to interact with the other compiler phases.

(2) Demonstrate that they are capable of generating good code in the context of an optimizing compiler.

To do this, we will take a dual approach. First, the basic requirements on the structure of a compiler will be discussed. Then, to show how especially high-quality code can be generated, we will use the structure of the PQCC compiler (section 1.2), which is patterned after Bliss-11 (Wulf et al [1975]). Both the PQCC compiler structure and the basic compiler requirements will be discussed in the next section.

3.2. The Compiler

3.2.1 Compiler Structure and TCOL

Several crucial decisions must be made with respect to the structure of the compiler.

One such decision must be made with respect to the language. In particular, we would like to parameterize the compiler by language as well as machine, so that a compiler for any language-machine combination can be generated. This thesis takes an UNCOL-like approach (UNiversal COmputer Language, Strong[1958]): the front end of the compiler translates a source language into an intermediate language, and the back end (code generator, etc.) translates the intermediate language into a machine language.

One problem with an UNCOL, which has limited its use in the past, is that an UNCOL notation must be at a sufficiently high level to avoid any assumptions about the underlying machine architecture, but at a low enough level to avoid any assumptions about the high-level languages to be translated. There are examples of UNCOLs that have made these assumptions in either the machine or language direction (Conway [1958], Orgass & Waite [1969]). The price of making such assumptions is that the corresponding translation becomes inefficient and/or complicated for non-conforming languages or machines. The language we will use, TCOL (Tree COmmon Language), alleviates this by using a very "low-level high-level" language. The notation is high-level in the sense that it is parse-tree-like to avoid any assumptions about the target instructions. It is low-level in the sense that assumptions about language constructs are avoided, by fully decomposing data accesses into index computations, fetches and stores, separating the description of the data types, and so on. The second problem with the original UNCOL approach is potential inefficiencies for special language or machine constructs. To avoid *this*, we allow the addition of new TCOL operators, by manually defining them in terms of existing ones, by extending the tables dependent on TCOL operators, or by defining new data types.

With these two guidelines (intermediate level, and extensibility), TCOL has proved to be quite useful in separating language-dependence and machine-dependence. The use of such intermediate languages in compilers will probably become prevalent in the near future.

The actual operators and definition of TCOL as used for this work are given in Appendix A. For an example of TCOL, the reader may wish to refer ahead to Figure 4, which shows a program and its TCOL representation. However, the details of the notation are not important for the understanding of this chapter.

Relatively little is required of the structure of the other components of a compiler using the code generation scheme described here. The principal requirement is a TCOL representation of the input program. The code generator produces output in a form that can be used in multiple ways, as we will see in section 3.2.4. Register and storage allocation can be performed either during or before code generation. The various compiler functions such as these, and their relation to the formalism, are discussed in the next three subsections.

As mentioned in the previous section, one compiler structure that could be used with the code generator described here is that of Bliss-11 (Wulf et al [1975]). Figure 3 illustrates this structure; the phases are as follows:

- LEXSYN: Lexical and syntactic analysis; the input to this phase is the source program text; the output is an abstract program tree (TCOL in our case).

- FLOW: Flow analysis; common sub-expressions are found, and sequencing of operations is rearranged.

- DELAY: Performs specialized optimizations on the tree, and also determines the "shape" of the ultimate code (e.g., where registers would be desirable).

- TNBIND: Allocates temporaries required by the program to the register types available.

- CODE: The actual code generation; input is rearranged/decorated parse tree, output is symbolic code stream. This is the concern of this thesis.

- FINAL: Performs peephole and branch optimizations that could not be recognized until actual code adjacencies are known.

There are also subphases of these compiler phases, but this will not be important for our purposes. What is important to note is that a careful balance is struck with respect to the ordering of compiler functions in this structure. The guiding principle is to perform a function where it is most advantageous in terms of the information available and the ease of description of the function. For example:

(1) DELAY determines the shape of the ultimate code to give TNBIND information about register requirements, allowing better global and local allocation of temporaries.

(2) CODE can then concentrate on the case analysis for code generation, given the registers that will be available for use.

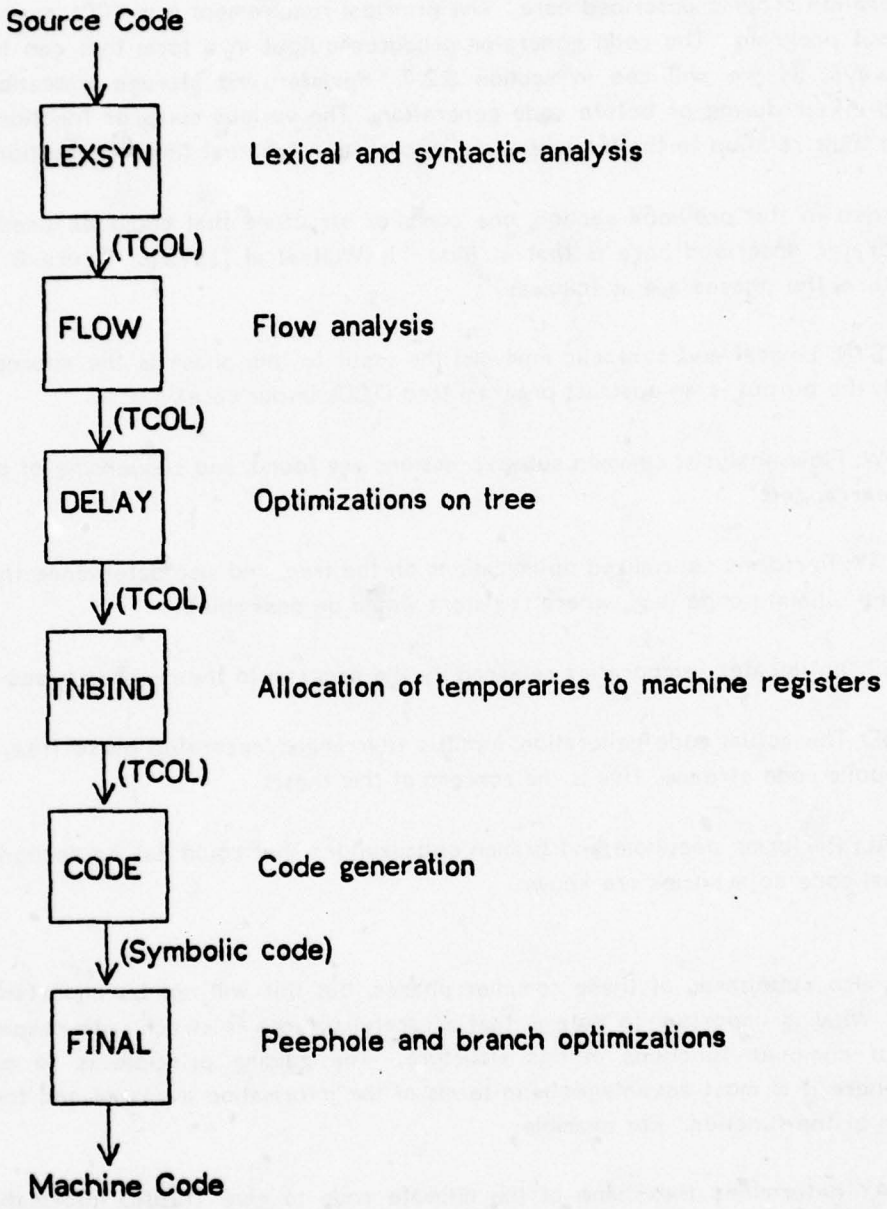


Figure 3: Structure of the Bliss-11 compiler, being used as basis for PQCC.

(3) The optimizations performed by FINAL could not be performed until the code is in symbolic instruction form.

3.2.2 Storage Allocation

The procedural information associated with a program is described with a TCOL tree. The declared properties of the data locations must also be expressed; this is normally done in a symbol table, and we will take this approach. However, to represent symbols machine-independently and language-independently presents more of a challenge. We will discuss how this can be done, although a full implementation has been left to further research.

A symbol table typically specifies information such as a print name, a data type (e.g., integer, real), an allocation type (e.g., own, dynamic), a size (e.g., for arrays or records), and other properties.

To separate the essence of the language- and machine- independent information about a symbol for the purposes of this thesis, two kinds of information are defined: an *access function* and an *allocation function*.

The *access function* is represented by a *location tree* for each symbol, which specifies the location the symbol represents. If the symbol is not a simple location, for example if it is an array or stack location, the location tree contains parameters that are filled in for any instantiation (use) of it in the program tree.[9] For example, a reference to a 10 by 10 array might be expressed as memory address A+$1+10*$2 where A is the base address of the array and $1 and $2 are the indices. A location tree thus specifies a (parameterized) access to a location.

An *allocation function* is required in order to create the symbol when it is declared, and possibly to release it when appropriate. Some examples are:

(1) A simple one-word own variable allocated at compile(load)-time

(2) A dynamic array, probably allocated by calls to run-time routines

(3) A stack local, requiring manipulation of a stack pointer

---

[9] There may also be some auxiliary code associated with every reference to a symbol, e.g., an array bounds check.

The allocation function and access function might be thought of as a secondary, language-independent symbol definition, the primary definition being that defined by the language directly, in terms of the language data types and data structures. The access function and allocation function are derived from the primary definition. The compiler's front end, which creates the TCOL representation, must know the machine data type (section 2.3.6) or virtual machine data type (section 3.2.5) for each symbol. This data type is derived from the language data type of this symbol, as will be described in section 3.2.5. Given the data type, the compiler knows:

(1) The amount of space required for this symbol, and (word boundary) alignment properties if any; together with the allocation type of the symbol this specifies the allocation function.

(2) The encoding/decoding algorithm to translate an ASCII representation of a constant of this type (e.g., 1.234) to the binary representation for this type; any constants can thus be encoded into binary before code generation occurs.

(3) How to translate source language operators such as "+" into the unambiguous TCOL operators which are defined in terms of machine data types.

To simplify the discussion of code generation, we will assume that all symbol references in the tree are expanded as if the access function were a macro. We will similarly ignore the processing of variable declarations, by assuming that a prepass has allocated symbols and inserted into the TCOL program tree any code necessary for dynamic allocation and other run-time maintenance.

3.2.3 Temporary Allocation

In addition to the data locations explicitly declared by the user program, temporary locations are required to hold intermediate results of expressions. These temporary symbols must also be dynamically allocated to locations on the target machine, typically registers or primary-memory locations.

In the classical compiler, this allocation is done "on the fly" during the generation of code. That is, when a temporary is required in the evaluation of an expression, a location of the required type is reserved by some scheme, and that location is used and marked available again after it is no longer needed.

More optimal allocation of temporaries can be performed if the allocation is performed as a

separate phase, preceding code generation; such a pass allows many possible assignments of registers to temporaries to be considered, evaluating their relative costs. This is the strategy taken in the Bliss-11 compiler. Johnsson [1975] extensively discusses this subject, and evaluates algorithms for assigning *lifetimes* to each temporary, and *packing* these temporaries into the available register types.

A global temporary allocator such as this can pass information to the code generator by modifying the TCOL tree. In an implementation, this can be done by setting a *target* field in the operator node in question to the desired location to be used for the result.

The proposed code generator is independent of the scheme used for temporary allocation. If the target field is set, the node is treated just as if the programmer specified that location for the result. If it is not set, the code generator calls a storage allocator provided elsewhere in the compiler, specifying the type of location desired, and generates code using what it is given.

3.2.4 Object code

The output of the code generation phases is a stream of tuples. Each tuple consists of a *format*, and a set of *fields*. The *format* specifies how the *fields* are to be output; it may specify:

(1) A *label* is to be generated; in this case, the field specifies a user- or compiler-generated label is to be defined at this location in the output code (for reference to this location elsewhere).

(2) A *constant* is to be output; in this case, the field is a binary word of data to be stored at the current location in the output.

(3) An *instruction* is to be output; in this case, the format specifies one of the *instruction formats* defined for the target machine (section 2.3.3), and the fields are to be assembled and output according to that format.

Fields are of two kinds: *constant* fields, and *parameter* fields. The value to be output for a *constant* field is given directly, while the value for a *parameter* field may be a variable or code label not yet defined. (The operation code for an instruction is a constant field; a field specifying an operand will normally be a parameter field.) The reader may recall these as the constants and parameters in M-op formats in section 2.3.7; formats and their use will be further described shortly (section 3.3).

The tuples output by the code generator can be output directly as binary code, or they may be passed as a symbolic instruction stream representation to another phase of the compiler, for example, a peephole optimizer.

Forward references in the tuples (for example, forward branches) must be resolved before binary code is generated. This cannot be done during code generation, as these values are not yet known, but the fields can easily be set in a second pass over the tuples, or, of course, assembly code can be output, in which case the fields remain in their symbolic form.

Considerable improvement in code quality can be obtained by a second pass over the code, and many people have worked on this problem. For example, branch optimizations can be performed that could not be detected during code generation because code adjacencies were not yet known. Hobbs [1978] discusses solutions to these problems in the context of PQCC. In general, any optimization or filling-in that requires knowledge of code size or adjacencies should be performed after code generation.

3.2.5 The Compiler-Writer's Virtual Machine

This thesis will not propose to *completely* automate the generation of code generators; this is not desirable. Certain implementation decisions, that have historically been made by the compiler designers, are still left to human discretion. These include:

(1) The mapping from language data-types to machine data-types. For example, "long reals" in the source language may be mapped onto the 64-bit floating point machine data-type. Or, a language data-type may not correspond to a target machine data-type, in which case a *virtual* data-type must be provided via run-time code for each operation on that data type. This could be simplified by providing a library of machine-independent TCOL run-time routines from which the compiler designer could select.

(2) Procedure linkage and parameter-passing. The implementation of call/return mechanisms for call-by-name, call-by-value, call-by-value/result, or call-by-reference, are defined by the compiler designer in terms of the lower-level TCOL primitives.

(3) Dynamic storage allocation. Closely related to the implementation of subroutines is the scheme, if any, for allocation of local variables to the routine. In a language such as Algol, code must be generated for block entry/exit to set up the locals on the stack.

(4) Operating system interface. Certain source language constructs, such as input/output requests, are expanded into operating system or runtime system calls.

In making these high-level decisions, we essentially define a virtual machine on top of the target machine, with certain operations defined in terms of in-line or out-of-line subroutines. We will call this the *Compiler-Writer's Virtual Machine (CWVM)*. In practice, these decisions can easily be inserted after automatic generation of the basic code generator, by defining new entries in the code-generation tables described in the next section. How this is done in an easily human-readable notation will be described in the next chapter.

3.3. Template Schemas

In this section we define the LOP, which specifies the code generation process, as was shown in Figure 1 (page 5). In the next two sections (3.4-3.5) the code generation algorithms which use the LOP are defined. Finally, in section 3.6, a complete code generation example is given to tie these together. The reader may prefer to skim sections 3.3-3.5, read the example in 3.6, and then re-read these sections to more easily understand the definitions and their motivation.

The basic unit of the LOP is a *template*. The essential parts of a template are a *tree pattern* (LHS), and a *result sequence* (RHS). The *result sequence* specifies code to generate, or further submatches to perform, when the *tree pattern* is found in the program tree. The templates are grouped into *schemas*, and the LOP consists of a set of these schemas.

The schemas represent different contexts in which code can be generated. Multiple schemas (for multiple contexts) are theoretically unnecessary, but a hierarchy of them is used for practical reasons. There might be schemas for contexts in which:

- A flow result is required (a conditional branch; see Wulf et al [1975]).
- A value result is required (integer, real, boolean, etc.).
- No result is desired (a "statement" tree).
- An addressing mode is to be selected.
- Subcases of an end-of-loop test are to be selected.

The remainder of this section gives a more precise definition of schemas, templates, patterns, and result sequences.

A schema is an ordered set of templates. A template consists of:

(1) a *pattern tree*

(2) a *result sequence*

(3) a *resource usage set*

A *pattern tree* is a tree whose nodes are:

- For non-leaf nodes, a language operator which this node is to match.

- For leaf nodes, a set of *access modes*, namely the access set of an *operand class*. These match classes of constants and locations that can be referred to in a single machine instruction, and are machine dependent (section 2.3.5). Parameters may be associated with the pattern leaf nodes for reference, as described in Appendix A.

The *resource usage set* specifies register allocation and cost information for this template; it will not be necessary to define it further for our purposes.

The *result sequence* is an ordered set of *code specifiers*. A code specifier consists of a *format*, as defined in section 3.2.4, and a list of *field specifiers*. The format determines the interpretation of the code specifier and its field specifier(s); specifically, a code specifier may represent:

(1) An instruction to be output; in this case, the format number gives the instruction format, and the field specifiers define the fields of the instruction. The field specifiers may be:

   (a) A parameter defined in the corresponding pattern tree (e.g., this would be used for an address field of an instruction).

   (b) A constant (to define a fixed-value field, e.g., an opcode).

   (c) A *match-tuple* consisting of a schema (name) and a parameter; this is used when the actual instruction fields are determined by an operand class; the fields depend on the actual program subtree involved (given by the parameter).

(2) A label to be output; in this case the (one) field specifier is a parameter corresponding to the label.

(3) A binary word to be output; in this case the (one) field specifier is its value (a constant or parameter).

(4) A match-tuple; in this case the field specifier is a parameter (representing a program subtree that matched the LHS pattern tree) which is recursively looked up in the template schemas to determine the code to generate. For example, this sub-matching scheme will be used to generate code for various sub-parts of an IF-THEN-ELSE: the conditional jump, the THEN-body, the ELSE-body.

Examples of patterns and result sequences will be given in figures 5-7, for the code generation example.

3.4. Code Generation Algorithms

It is the intent of the template schemas to describe the translation from TCOL to a machine language. However, the template schemas are meaningless without defining the algorithm which uses them; the algorithm is the topic of this section.

The reader may already have inferred the general idea of the use of templates at this point. The template patterns are matched against the TCOL program tree and the corresponding result sequence specifications are processed. With the templates as defined, the only degree of freedom we will have in using different algorithms is in the interface "between" the templates, for those templates which compute arithmetic expressions where intermediate values must be stored. That is, the templates must be composed recursively to match an entire program tree.

Recall that the leaves of a template are access sets; the operands must be in these specified locations (access modes) in order for the template to be applied. If a leaf of a template does not match the corresponding segment of the program tree, it is possible to make the template applicable by performing the subtask of a store into an allocated location (access mode) of the type required by the template operand. Or, if it is the *destination* of the template that mismatches the program subtree, it is possible to make the template applicable by allocating a location of the type of the template destination, and following the template result code by a store into the desired location from the allocated location. This process of making the template match is called *subtargetting*. Subtargetting is used, for example, when an instruction requires one or more of its operands in registers, but the operands in the program tree are not in registers. Note that there are two types of subtargetting, depending

on whether the offending access mode is the destination or source in the template.[10]

Perhaps the most obvious way to apply the templates is the exhaustive "brute force" approach. Given a program tree, there may be a number of templates which match the top of the tree, ignoring mismatch of the access set leaves, which can be subtargetted. For each of these templates, there may be a number of access modes which would satisfy the operand constraints, to which the operands could be subtargetted. Recursively, for each subtargetting subtask, there may be a number of templates which match, and so on. In an exhaustive algorithm, we would recursively try all these possibilities at each node, and use the match(es) that gave the least expensive code sequence.

The exhaustive approach here is not new. Newcomer[1975] and Aho & Johnson [1976] use approximately this algorithm. Aho & Johnson show that the time for the exhaustive algorithm is linear in the number of tree nodes, and exponential in the number of choices (instructions, or M-ops and access modes) at each point.

It should be noted in passing that Aho & Johnson's assumptions (and Newcomer's) do not correspond to ours, although the complexity result still applies. For example, we subdivide the selection of instructions into the selection of an M-op and the selection of an access mode for each operand, to reduce the number of patterns (choices) to deal with. We also deal with control constructs, rather than just arithmetic, and allow M-ops to consist of multiple actions (more than one store) or to compute arbitrary expressions rather than corresponding to a single language operator. Some of these differences add to the complexity of code generation (e.g., multiple-operator M-ops), and others to the complexity of code generator generation (e.g., multiple-action M-ops).

The time required for the exhaustive approach might not be excessive for many machines, since the number of alternatives possible at each node is often small, and the size of arithmetic expressions is also empirically small (Knuth [1971]). However, another alternative deserves consideration:

We might consider <u>ordering</u> the alternatives, which in our case corresponds to ordering the templates in a schema, and using an algorithm which selects the <u>first</u> template which matches a given program node. The instructions would be ordered so that less expensive special cases occur before general cases; for example X←X+1 would occur before X←X+constant. We order both the M-op alternatives (for the instruction) and access mode alternatives (for each operand class) in this way: the pattern templates are sorted by increasing cost per number

---

[10] These will correspond to the Fetch/Store Decomposition Rule in the next chapter.

of nodes. This results in an algorithm which "bites off" the largest possible subtree at the current program tree node at each step, and subtargets the remainder. We will consequently refer to this as the "Maximal Munching Method" (MMM).

Other alternatives to the MMM algorithm and the exhaustive algorithm are possible. Rather than picking the largest possible access mode matching at each point, we could always use some common denominator for temporary results, such as a general purpose register, which satisfies all the operand classes. This is still simpler than the MMM algorithm, but will not generate as good of code in general.

Experimentation with various code generation algorithms and their relative performance is beyond the scope of this work. However, small test cases suggest that the MMM algorithm does nearly as well as the (optimal) brute force method, and at much less cost. This is therefore the method used here.

3.5. The MMM Algorithm

In figure 2, a simplified algorithmic version of the MMM algorithm is given.

In order to understand this algorithm, it is first necessary to understand the data structures, given first in the figure. Associated with each pattern leaf (except for "closed constant" leaves, e.g., "2") are:

(1) An operand class "OC" which the pattern demands in this leaf position of the pattern tree.

(2) A parameter "parm" (implemented in the figure as an integer index into an array) which is to be associated with the program tree node which matches this pattern node. Parameters are used to save the subtrees which match pattern leaves for later use in subtargetting and in generating code.

For the purposes of subtargetting a program subtree, we need to know:

(1) The pattern leaf the subtree matched, to determine the set of access modes this node must match. A pointer to the pattern leaf is saved in the "pleaf" field of the subtree root node.

(2) Whether the subtree matches one of the access modes specified by the pattern leaf.

```
class pattern = patternleaf or patternnode;
patternnode= (integer op {TCOL operator #}; tree array sons);
patternleaf= (integer op; operandclass OC; integer parm; integer value);
class tree = (integer op; tree array sons; ... boolean match; tree pleaf);
class production = (pattern LHS; resultsequence RHS);

boolean procedure Treematch(tree T; pattern P; tree array Parms);
  {matches tree T against pattern P, saves parameters in Parms}
  if P.op=constant then return(T.op=constant and T.value=P.value)
  else if P.op=location then
    begin {leaf (location)}
    Parms[P.parm]←T; T.pleaf←P;
    if T ∈ P {match OC} then T.match←TRUE else T.match←FALSE;
    return(TRUE);
    end
  else
    begin {non-leaf}
    if P.op≠T.op or size(P.sons)≠size(T.sons) then return(FALSE);
    for i←1 thru size(P.sons) do
      if not Treematch(T.sons[i],P.sons[i],Parms) then return(FALSE);
    return(TRUE);
    end;

procedure GenCode(tree T);
  {generates code for TCOL tree T}
  begin
  Pset←FindProds(T); {get set of templates to try for this tree}
  foreach P∈Pset do
    begin Mset←NIL array; if Treematch(T,P.LHS,Mset) then exitloop end;
  {Fetch subtargetting}
  foreach M in array Mset do
    if not M.match then
      begin Allocate(M.pnode); GenCode( M.pnode "←" M ) end;
  {Generation of code for RHS of template}
  DoResultSequence(P.RHS,Mset);
  {DeAllocate}
  foreach M in array Mset do
    if not M.match then
      Deallocate(M.pleaf);
  end;
```

Figure 2: A simplified version of the MMM code generation algorithm. Comments are in {...}. "X.F" means F field of variable X; fields are defined in the class definitions at the top.

If so, the "match" field of the subtree root node is TRUE, otherwise it is FALSE, and subtargetting must be applied.

The tree and pattern node fields are shown in the figure. Following the data structure definitions are the two central routines, TREEMATCH and GENCODE.

TREEMATCH takes a pattern and tree as argument, and returns TRUE if they match or can be made to match by subtargetting. The array PARMS contains pointers to tree nodes that were matched against pattern leaves; those that did not match must be subtargetted. Those that match as well as those that must be made to match are returned in PARMS because both will be needed in the generation of code for the right-hand side (RHS) of the template.

GENCODE uses a hashing scheme (FINDPRODS) to find a set of templates which might match the program tree, uses TREEMATCH to find the first of these templates that can be made to match the program tree, subtargets where necessary, generates code for the template itself, and then cleans up by deallocating the temporary locations used (if temporary allocation is done *before* code generation, allocation and deallocation has been handled in an earlier phase).

The complete algorithm is several pages of code; this is an oversimplification. However, the remaining details are relatively straightforward. In particular, we have not given the details of the processing of the result sequences (this should be clear from the result sequence definition and the example), or the *store* subtargetting (this is similar to fetch subtargetting, but a special check is made for the destination of "←"s for mismatches).

3.6. Example

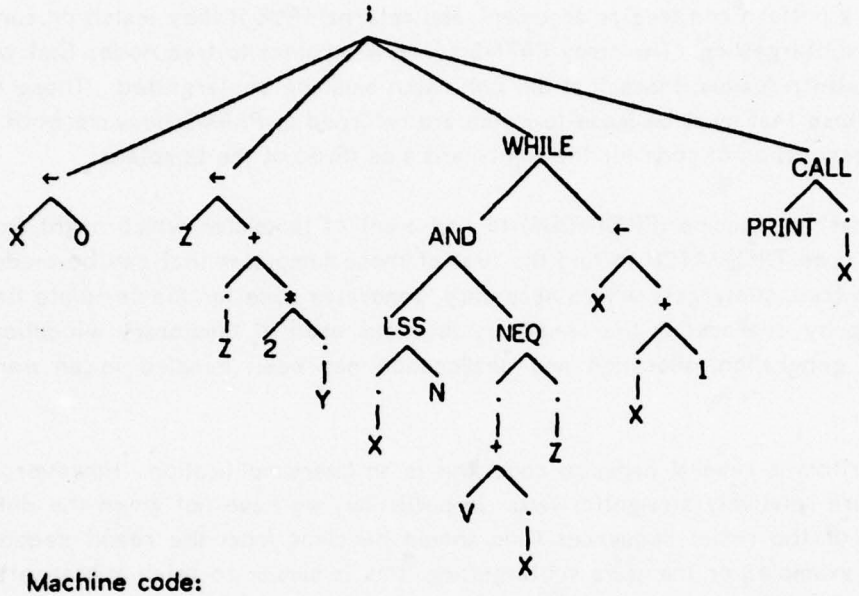
We will demonstrate the use of template schemas by tracing through the generation of code for a small program using the MMM algorithm.

Figure 4 shows an example program and how it would be internally represented in TCOL. We will show how the CODE phase would generate PDP-11 machine code for this example using the MMM algorithm with the tables (template schemas) in Figures 5 through 7.

The notation used in Figures 5 through 7 requires some explanation. Each figure is a schema; the figures show the schemas for the statement context, value context, and flow context,

```
X←0;
Z←Z+2*Y;
WHILE X<N AND V[X]≠Z DO
   X←X+1;
PRINT(X);
```

After conversion to internal TCOL tree:



Machine code:

```
      CLR R1
      MOV Y,R2
      ASL R2
      ADD R2,Z
      BR L01
L02:  INC R1
L01:  CMP R1,#N
      BGE L03
      CMPB V(R1),Z
      BNE L02
L03:  MOV R1,-(SP)
      JSR PC,PRINT
```

Figure 4: Example program. X, Y, and Z are integers; N is a constant; V is a byte array.

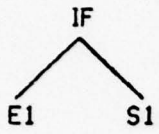
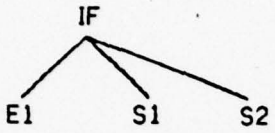
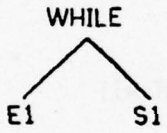
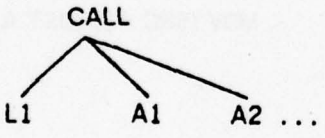
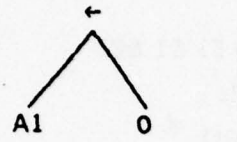
Pattern	Result Sequence
	(→ E1 &1 &2) &1: (S1) &2:
	(→ E1 &1 &2) &1: (S1) BR (ADR &3) &2: (S2) &3:
	BR (ADR &2) &1: (S1) &2: (→ E1 &1 &3) &3:
	%REPEAT 1 MOV (SRC A%I) "-(SP)" %END JSR "PC" (DST L1)

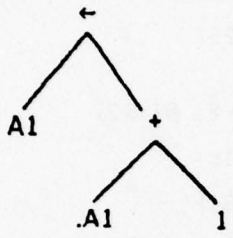
Figure 5: Statement context schema of LOP (top operator has no value)

Pattern

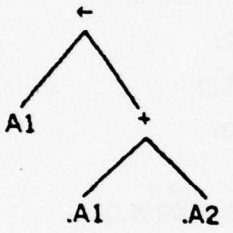
Result Sequence



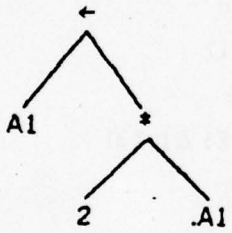
CLR (DST A1)



INC (DST A1)

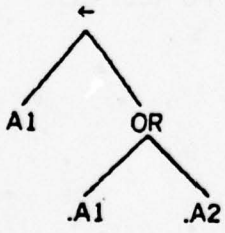


ADD (SRC A2) (DST A1)



ASL (DST A1)

MOV (SRC A2) (DST A1)

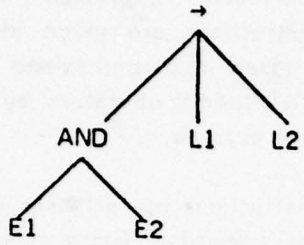


BIS (SRC A2) (DST A1)

Figure 6: Value-context schema of LOP (top operator "←").

Pattern

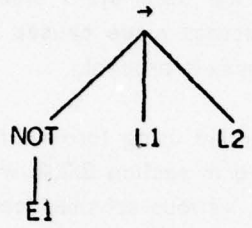
Result Sequence



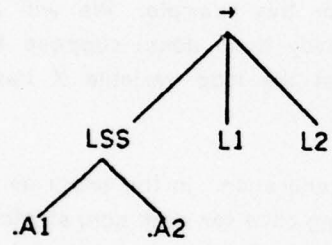
(→ E1 &1 L2)

&1:

(→ E2 L1 L2)



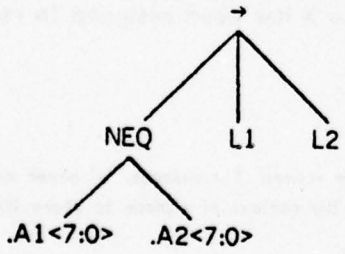
(→ E1 L2 L1)



CMP (SRC A1) (SRC A2)

BLT (ADR L1)

BR (ADR L2)



CMPB (SRC A1) (SRC A2)

BNE (ADR L1)

BR (ADR L2)

Figure 7: Flow-context schema of LOP (top operator is "→"). The semantics of the "→" nodes are that control goes to location L1 if the first son is true, else to L2. A single language operator corresponds to different TCOL operators according to the type and size of the data; for simplicity in the figures, only the 16-bit integer versions are shown (except NEQ, which illustrates an 8-bit compare).

respectively.[11] The leaves of the patterns are marked with S, E; A, or L, depending on whether they match a statement, an expression, an "addressable" expression, or a label expression, respectively.[12] In the result sequences, instructions are given in symbolic assembly-language form for readability; submatches are given in parenthesized form, i.e., "(context son_1 son_2 ... son_n)". Labels are preceded by "&", special operators by "%". The use of these should become clear as we proceed through the example.

The reader may note that the submatches for fields of instructions use schemas (DST, SRC, ADR) for operand classes, that we have omitted here for simplicity. These schemas have patterns for each access mode that can be matched by the parameters (subtrees) passed them, and the corresponding result sequence for each access mode causes output of the appropriate PDP-11 addressing mode, register field, and index if present.

In an implementation, the result sequences would be encoded using formats to indicate the types of code specifiers and field specifiers, as suggested in section 3.2.4; we are omitting implementation details here for readability. For example, various schemes can be used for representing and indexing pattern trees for efficient and flexible use.

Temporary register allocation will be done on the fly for this example. We will assume simple global allocation of variables to locations has already been done: suppose that all locations but X have been assigned to memory, and that the loop variable X has been assigned to a register.

We perform a traversal of the program tree in the code generation. In the program tree in figure 4 the top operator is ";", so we proceed by generating code for each son, starting with X←0.[13] This tree is matched by the first template in Figure 6; the corresponding result sequence indicates the generation of an instruction with opcode field CLR and destination field given by looking up X in the DST table. Let's suppose X has been assigned to register

---

[11] Note that we only need these three contexts which represent complete actions. For example, "+" never occurs at the top of a template pattern tree because it makes no sense except in the context of a place to store the result (value context).

[12] The letters are used as a reminder of what the parameters are for readability; the pattern leaves in actuality are either access sets or "wild" (i.e., match anything, e.g., E and S). A is the access set of both DST and SRC; L is the access set of ADR

[13] The semantics of ";" could be built into the code generation algorithm, simply causing a recursive call on the sons.

1, so that "CLR R1" is the instruction generated.[14]

We then proceed to the second son of ";", and find that the first match is the third in Figure 6: A1←.A1+.A2. A1 is matched exactly by the Z leaf, but 2*.Y will require a temporary location and another match; *subtargetting* occurs. CODE allocates a temporary location, say R2;[15] then it calls itself recursively to match R2←2*.Y. This is matched by A1←2*.A1, but again requires a further submatch, R2←.Y, this time because the parameter A1 is used twice in the pattern tree, and since the first occurrence of A1 matched R2, Y must be in R2 as well. R2←.Y matches the 5th template in Figure 6 exactly, resulting in the generation of MOV Y,R2. Upon returning from the two recursive calls, CODE generates:

```
ASL R2
ADD R2,Z
```

The WHILE is then matched, by the third template in Figure 5. Note that CODE "understands" the generation and use of labels, which are indicated in the result sequence as &1, &2, etc. If symbolic output were being generated, names would be generated here. For example, the first two code specifiers would generate BR L01 and a label definition L02. The next line causes a recursive call to match X←.X+1, and INC R1 is generated. Then L01 is defined.

The next code specifier causes lookup of the AND subtree in the flow-context table. The AND is matched and indicates a submatch on the first son, "LSS", in the same table. This match generates

```
CMP R1,#N
BLT L04
BR L03
```

Another label is generated, then the second son is matched, generating

---

[14] Note that something as complicated as "@5(R3)" (in PDP-11 assembler notation) could have matched the access set here, we just happened to have simply a register.

[15] In the MMM algorithm, we pick the best (cheapest per node) access mode for the operand class (DST in this case) which can be made to match (A1). But since the top operator of the subtree is "*", the first one found is register mode (if a register is available).

```
L04: CMPB V(R1),Z
     BNE L02
     BR L03
```

Code then returns to complete evaluation of the WHILE node by generating L03:.

Finally, the function call is processed. A simple argument passing mechanism is assumed. This last son also serves to illustrate how CODE could be made to handle nodes with an arbitrary number of sons, with a special control command in the result sequence, %REPEAT. The code generated is:

```
MOV R1,-(SP)
JSR PC,PRINT
```

This completes the example. An additional pass over the code stream is used[16] to resolve forward references, deal with long vs. short jumps, and remove redundant branches. The bottom of Figure 4 shows the complete code sequence for the example after removal of two redundant branches.

The main purpose of the example and this chapter is to design and demonstrate enough of a feasible code generator to make the work in the next chapter possible. That is, we need a model of code generators to generate code generators. There are many issues in the implementation of a code generator we cannot hope to discuss in detail here. There are several ways to represent result sequences, as pointed out earlier. There are more complexities involved in dealing with operand classes that we have avoided for simplicity; for example, there are both byte and word addressing modes on the PDP-11, necessitating separate templates for byte and word instructions. There are issues with respect to the TCOL representation; for example it might be better to decompose the CALL node used here into operations such as PUSH on the stack to make the calling and parameter-passing mechanism explicit in the TCOL. Another likely change to the notation would be to encode the explicit store nodes ("←") as a target field in the expression nodes (e.g., "+") which gives the location to be used for the result (as a location tree). Finally, an indexing scheme for the templates in a schema is essential to avoid serial matching of all the patterns. A simple hash by the primary operator (1st son for "→", 2nd son for "←", main operator for statement

---

[16] This could be implemented as a separate FINAL-like pass (Wulf et al[1975]) and/or concurrently with CODE as part of the instruction EMIT routine called for each n-tuple.

context) worked well in the prototype implementation described in section 3.8. The template matching might be implemented even more efficiently by combining the patterns (automatically) into a single match tree, as done by Weingart [1973]. Specifically, this approach is best when the patterns are complicated (many nodes), and slightly differing patterns occur. It should also be noted that the representations described in this chapter do not necessarily correspond to those currently planned by PQCC. For example, the contexts described in section 3.3 may be represented by special flags in the tree nodes, and the control constructs (e.g., WHILE-DO) may be decomposed into blocks of code and conditional jumps (flow contexts).

3.7. Use in a Compiler

The previous section described how the template schemas could be used in a stand-alone code generator. Better machine code can be generated if we use the template schemas in multiple phases, for example if we perform temporary allocation as a separate phase, so that many assignments of temporaries to locations can be considered.

In the Bliss-11 compiler structure (section 3.2.1), the template schemas could be used in the DELAY, TNBIND, and CODE phases. Basically:

(1) The DELAY phase would act as a pseudo-code-generator. That is, it would perform the code generation algorithm described in the previous section without generating code, but only to determine where locations of each storage base type are required (namely, where subtargetting is performed). We could do this by assuming an infinite number of allocatable registers of each type, and setting the target fields of tree nodes which require registers.

(2) The TNBIND phase would then allocate temporaries as best it can to the nodes of the program tree which have been marked as requiring them. TNBIND knows the actual number of locations of each storage base type, and the lifetimes (periods of use) of the temporaries which are to be stored in these locations.

(3) The CODE phase would then generate code. Unlike in stand-alone mode, the targets for the temporary results will have been specified, so that no on-the-fly allocation will be required. Alternatively, we could permit TNBIND to fail to assign a register of the type required to make a node match a template exactly; to handle this, CODE would fall back into "on-the-fly" mode to subtarget to an allocated temporary (taking into account TNBIND's assumptions about register use; typically, a memory location would be used in place of a register).

3.8. An Implementation

In order to test the model of code generation proposed in this chapter, a prototype code generator was implemented. As mentioned earlier, the goal of this test of the model was to set the stage for the next chapter, so only a minimal system was necessary: in fact, the code generation algorithm used is independent of the the LOP table and the discussion of its generation in the next chapter.

Because PQCC's compiler is not yet designed and built, it was necessary to build a stand-alone code generator for TCOL with makeshift simulations of register and storage allocation (TNBIND) and code output (FINAL). The interfaces to these two compiler functions are quite simple. We assume FINAL provides a routine which takes a list of symbolic fields and a format specifying how to assemble them into an instruction or binary constant; this routine could output code directly or create a data structure. TNBIND can communicate with CODE through modification of target fields in the nodes, and by supplying routines to process storage declarations and allocation of registers of various kinds.

The structure of the prototype code generator and the functions of its main routines are shown in Figure 8. An example run of the code generator is shown in Appendix C.

No serious problems with the code generation model or its implementation were encountered in the prototype. It is currently planned that a code generator based on the prototype be integrated into the compiler system being built by the PQCC project.

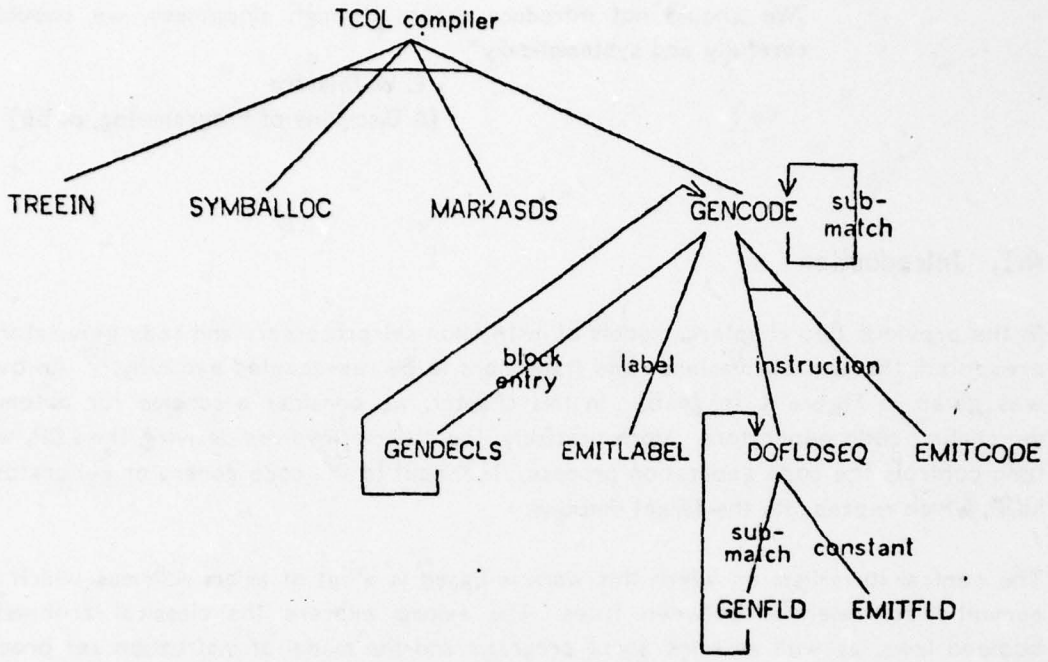


Figure 8: Structure of prototype code generator. The named procedures call one of the subprocedures shown, or all subprocedures when marked by horizontal line through alternatives. TREEIN inputs TCOL, SYMBALLOC outputs space reservations for symbols (eg, own variables), MARKASDS marks the symbol nodes in the tree with the access mode (AM) in which they fall, and GENCODE generates the code, being called with a reference to the top of the program tree as parameter (initially). After finding a template which matches the node it is given, GENCODE interprets the corresponding result sequence, which may specify a recursive submatch of another node (recursive GENCODE call), output of a label (EMITLABEL), local declarations to be processed (GENDECLS called until code encountered again), or output of an instruction. In the latter case, DOFLDSEQ is called to interpret the field sequence; then the resulting fields are output by EMITCODE. Each field in the sequence may be specified directly (EMITFLD called) or determined by a submatch (for OCs), in which case GENFLD calls DOFLDSEQ recursively.

4. Automatic Derivation of Translators

"We should not introduce errors through sloppiness, we should do it carefully and systematically"

- E. W. Dijkstra
[A Discipline of Programming, p. 56]

4.1. Introduction

In the previous two chapters, models of instruction set processors and code generators were presented; these allow machines and translators to be represented explicitly.[17] An overview was given in Figure 1 (page 5). In this chapter, we consider a scheme for automatically generating code generators. More precisely, the scheme involves deriving the LOP, which in turn controls the code generation process. The input to the code generator generator is the MOP, which represents the target machine.

The central formalism on which this work is based is a set of axiom schemas which specify semantic equivalences between trees. The axioms express the classical arithmetic and boolean laws, as well as rules about programs and the model of instruction set processors. The axioms will be used to specify legal (semantics-preserving) tree transformations, and are used in a heuristic search for optimal code sequences. This search algorithm is the central result of the chapter. The tree equivalence axioms are presented in the next section (4.2); the search algorithm which uses them is presented in the following section (4.3).

The search algorithm is essentially a machine-independent code generator, which takes as input not only the source tree, but also a description of the target machine, namely the MOP. The algorithm uses the axioms to find a code sequence for the source tree in the target machine language. This machine-independent code generator could be used directly as the code generation phase of the compiler (i.e., the MOP could be used directly, obviating the LOP). In practice, however, it is preferable to separate compile-time from compiler-compile time, to make the code generator more compact and efficient (heuristic searches tend not to be!). This is done by the introduction of the LOP table, as shown in Figure 9. Recall from the previous chapter that the LOP contains *templates* used by the code generator. Each template

---

[17] Note that TCOL provides a common representation for instruction actions, the source language, and the axioms used to link the two. This is not strictly necessary, since translations between notations could be done, but it makes the overall organization conceptually simpler.

consists of a tree pattern, and a code sequence to generate when that tree occurs in a source program. The algorithm labelled SELECT in the figure is used to select the pattern trees, call the machine-independent code generator to find the corresponding code sequences, and enter the resulting tree/code-sequence pairs (the templates) into the LOP. The SELECT algorithm, and associated problems in the generation of the LOP table, are the subject of section 4.4.

In section 4.5, the relation of this work to other work in the area will be discussed. Finally, in section 4.6, examples and an overview of the actual implementation will be given.

4.2. Tree Equivalence Axioms

4.2.1 Overview

The central basis for the work in this chapter is a set of axioms specifying equivalence of programs. Examples of some of these axioms are shown in Figure 10.

Note that the axioms cover a wide class of equivalences: arithmetic and boolean laws, rules about storage and side effects, and rules about program sequencing (including the semantics of the program counter, PC). The axioms define an "algebra of trees" which will provide the search space for our problem. In the remainder of this section (4.2), the various flavors of axioms are discussed in more detail.[18]

4.2.2 Arithmetic and Boolean Laws

The arithmetic and boolean axioms are relatively straightforward; see Figure 10 for examples. Boolean laws include commutativity of AND and OR, DeMorgan's law, and the double-complement rule. Arithmetic axioms include commutativity of addition and multiplication, special cases of adding, subtracting, or multiplying by zero or one, relations between addition and subtraction (e.g., a-b=a+(-b), -a=0-a), and so on.

---

[18] The axioms are represented in the implementation as productions whose left-hand side (LHS) and right-hand side (RHS) are parenthesized TCOL trees; these axioms are shown in Appendix F. Note that the axioms in Figure 10 are shown as bidirectional, i.e., "A ↔ B" may be applied to transform A into B or vice versa. The actual axioms in Appendix F are unidirectional (LHS into RHS). This difference is not crucial, it simply allows additional control over the search. (Three of the axioms actually used are represented in the code rather than the parenthesized notation; this is also due to implementation considerations.)

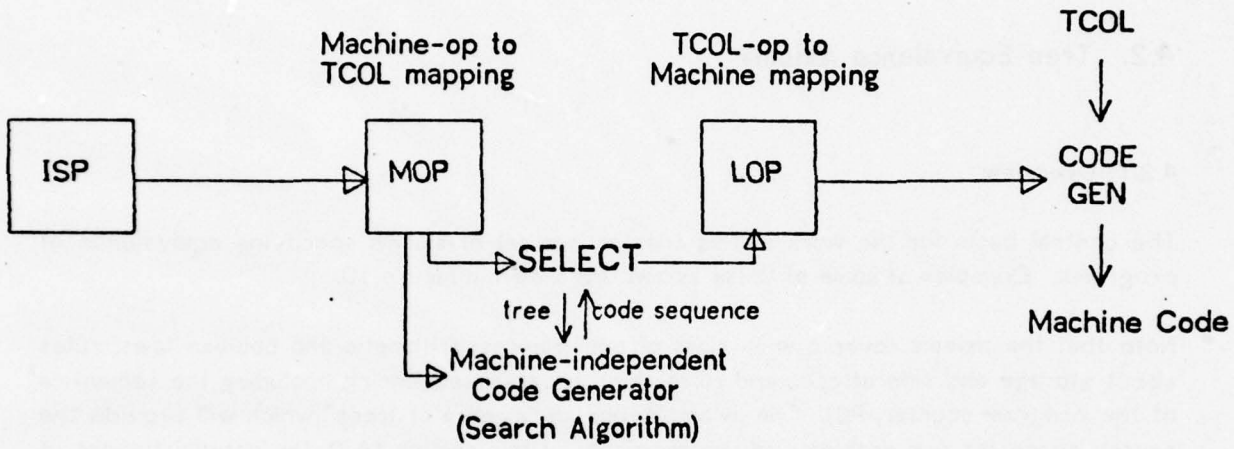


Figure 9: Highly simplified view of code generator generator, generating LOP from MOP. SELECT insures that the necessary TCOL to code sequence mappings (productions) are in the LOP, calling the search routines to find code sequences for the necessary trees. The search routines are expanded in figure 11.

Figure 10: Tree Equivalence Rules (Examples)

Boolean axioms
not not E <=> E
E_1 and E_2 <=> not((not E_1) or (not E_2))
E_1 and E_2 <=> E_2 and E_1
E and E <=> E

Arithmetic axioms
E+0 <=> E
-E <=> 0-E
E_1*E_2 <=> E_2*E_1
-E <=> (not E)+1
E↑1 <=> E*2

Relational axioms
not (E_1 geq E_2) <=> (E_1 lss E_2)
(E_1 lss E_2) or (E_1 eql E_2) <=> E_1 leq E_2

Fetch/Store decomposition rules
E_1(E_2) <=> S←E_2; E_1(S)
S_1←E <=> S_2←E; S_1←S_2

Side-effect compensation axioms
S; D←E <=> S if D is temporary-type SB
S; D←E <=> Alloc(D); S if D is general-type SB

Implementation rules
if E then S <=> if (not E) then goto L; S; L:
if E then S_1 else S_2 <=> if E then goto L_1; S_2; goto L_2; L_1: S_1; L_2:
while E do S <=> L_1: if not E then goto L_2; S; goto L_1; L_2:

Sequencing Semantics axioms
goto E <=> PC←E
PC←PC+n; S<space n> <=> <nil>
if E then S <=> if not E then PC←PC+n; S<space n>
goto L_1 <=> goto L_1-L_2+PC; L_2:

Notation:
L: location in instruction store;
D: location in data store;
E: combinatorial tree;
S: statement (assignment or conditional) tree;

Axioms are required for each data type (section 2.3.6). A special set of axioms specify properties of the data type representation. For example, for two's complement integer arithmetic:

```
-E   <=>  (NOT E)+1
E↑1  <=>  2*E    ("↑" denotes shift left)
```

Other axioms simply specify relationships between relational and boolean operators. In the implementation, special simple cases of arithmetic are also encoded as axioms, e.g., 1+1=2, since so few cases are required. These could of course be obviated by making the program "smarter".

4.2.3 Fetch/Store Decomposition

```
E1(E2)   <=>  S←E2; E1(S)          (Fetch Decomposition)
S1←E     <=>  S2←E; S1←S2          (Store Decomposition)
```

These rules are centrally important. They represent the fact that there are storage locations on a machine, and that they can be used to hold intermediate results.

Fetch Decomposition states that an expression E_1 with a sub-expression E_2 can be computed by first computing E_2 in a location S, and then replacing E_2 with S in the computation of E_1. Note that the location S must be available for use to hold the intermediate result, and S may not be stored into for any other use until after the fetch of S in the computation of E_1. This is the only axiom with side effects on the machine state. Aspects of the machine state will be discussed in section 4.4.2.

The second rule, Store Decomposition, is a special case of Fetch Decomposition in which E_1 is an assignment statement. It is treated separately because of the way the means-ends analysis works, in section 4.3.2.

4.2.4 Side Effects

```
S; D←E   <=>  S   [if D is temporary-type]
S; D←E   <=>  Alloc(D); S   [if D is general-type]
```

If a machine-operation tree has more than one effect, it may be possible to use the instruction for a subset of its effects, and ignore or compensate for the undesired side effects. The side effects we are concerned with here are in the form of changes (or conditional changes) to the value of a storage location on the machine. The storage bases on the machine were classified in section 2.3.1 according to whether they were temporary, general-purpose, or reserved. *Temporary locations*, for example, condition codes, may be destroyed in the process of generating code. *General purpose locations*, for example registers or memory locations, may be used if they are allocated. If no location of the type required is available for use, this could involve saving the value in another location temporarily. *Reserved locations*, for example a stack pointer or subroutine linkage register, may never be used for intermediate results.

The use of these rules about side effects will be discussed further in section 4.3.4. Side effects on the program counter, the 4th type of storage base, will be dealt with in the next section.

4.2.5 Sequencing Semantics

The remaining axioms are concerned with flow of control. We assumed, in the model of instruction set processors in Chapter 2, that the processor state contains a program counter, PC. The PC's value, after each instruction cycle, is the address of the next instruction to be executed.

Sequencing semantics are specified by two kinds of axioms: First, there are definitions of higher-level constructs in terms of conditional and unconditional jumps. For example:

```
if E then S  <=>  if not E then goto L; S; L:
```

Second, there are three basic axioms which specify the semantics of the program counter. These will be referred to as the *Hop, Skip, and Jump rules*. The *Hop rule* merely defines the PC:

```
goto L  <=>  PC←L
```

The *Skip rule* says that a block of code is skipped when the PC is incremented:

```
PC←PC+n; S<space n>  <=>  <nil>
```

The "S<space n>" in this axiom represents a tree S which requires exactly n *words* (section 2.3.1) of machine code. The Skip rule will be combined with the definition of "if" in the examples in the next section for the special case of *skip-decomposition*:

if E then S <=> if not E then PC←PC+n; S<space n>

We will also use the abbreviation

E->L for if E then PC←L

in the next section, using the operator "->" for the very common case of a conditional jump.

Finally, the *Jump rule* allows the use of relative and absolute jumps interchangeably:

$PC \leftarrow L_1$ <=> $PC \leftarrow PC+L_1-L_2$; L_2:

4.3. A Search Algorithm using Tree Transformations

4.3.1 Introduction

Now let's consider the machine-independent code generation problem. We are given a machine M with instructions m_1, m_2, ... m_n, and a goal tree G for which we would like to generate code (in the machine language of M). That is, we would like to find a sequence of machine-operation tree instantiations which is semantically equivalent to the goal tree G. The axioms presented in the previous section define the term "equivalent": if a subtree matches one side of an axiom schema, the subtree may be replaced by the instantiation of the other side. In this way, the goal G can be successively transformed into other trees, until eventually we may arrive at a tree which is a sequence of M-op trees:

$$G \Rightarrow G' \Rightarrow G'' \Rightarrow \ldots \Rightarrow m_{i_1}; m_{i_2}; \ldots m_{i_k}.$$

Because more than one axiom may be applicable to a tree at any point, and we can test for the termination condition of a sequence of M-ops, we have a classical search problem. That is, starting with G, we may apply all applicable axioms to obtain a set of equivalent trees, recursively apply all applicable axioms to *those* trees, and so on, until we have one or more instruction sequences for the goal tree.

Applying this algorithm literally is undesirable, as the search space is combinatorially large. Note that axioms may be applicable at more than one point in a goal tree, and more than one axiom may be applicable at each one of these points.

To deal with this problem, we will use some established methodology from the field of artificial intelligence. In fact, we will use not one method, but several, allowing the strongest applicable method to be used for each kind of information. To do this, the axioms have been divided into three classes:

(1) Transformations. These are the axioms concerned with arithmetic and boolean equivalence. Transformations will be used in conjunction with *means-ends analysis* in section 4.3.2.

(2) Decompositions. These axioms are normally those concerned with control constructs; they decompose constructs into sequences of other constructs, allowing the search to recursively proceed on subgoals. Decompositions will be used in conjunction with a general heuristic search described in section 4.3.3.

(3) Compensations. These are the axioms concerned with side effects. No search at all will be associated with these axioms; it will be possible to use them in a pre-pass on the MOP table, as will be described in section 4.3.4.

The use of these three kinds of axioms will be discussed in the next three sections. The reader may wish to refer ahead to Figure 11 (page 72) at this point, to get an overview of the routines involved in this process. Briefly, TRANSFORM applies transformations, SEARCH applies decompositions, and INDEX applies compensations. Note the recursive relationship between SEARCH and TRANSFORM; SEARCH attempts to match a goal tree against the available machine-operations and decompositions, and then calls TRANSFORM to try transformations. The recursive calls of SEARCH and TRANSFORM are to process subgoals. All possible code sequences found for a given goal tree are returned by the search routines, and the best of these is chosen by SELECT to be entered into the LOP table. The "best" cost is determined by a user-supplied function of time and space; the time and space cost for instructions are known from the machine description, as described in Chapter 2. Following discussion and examples of the search algorithms, we will see how the algorithms are combined in the composite search for code sequences, and the figure will be explained in more detail.

4.3.2 Transformations

The first and "strongest" method we will use in the search for code sequences is means-ends analysis (Ernst & Newell[1969], Newcomer[1975]). Briefly, means-ends analysis is deciding how to get from what you have to what you want by representing the difference between the two in some way, and picking an action to perform on the basis of that difference, with the idea of reducing the difference.

In order to apply means-ends analysis we need both a starting point and a goal. However, in the search for code sequences, we have only a goal: the tree to be coded. Therefore, the algorithm involves two steps: first finding feasible instructions to work toward from the current goal tree, and then applying means-ends analysis to transform the goal tree to a form to which each such instruction is applicable:

(1) (FINDFEASIBLES) Using some simple heuristics, we find a subset of the M-ops, ordered by decreasing feasibility for use in implementing all or part of the goal tree. The heuristics are:

   (a) Pick instructions whose *primary operator* is the same as the primary operator of the goal tree. The *primary operator* of a tree is simply a convenient key used for indexing trees. Operationally defined, it happens to be the root operator of the second operand (the source) in the case of "←", and the root operator of the condition in the case of "->" or "IF"; thus the primary operator provides an index to the main expression computed by an instruction. This primary operator key is used frequently in this work as an indexing scheme to avoid serially matching trees.

   (b) Include (as second choice to the above) instructions whose primary operator is closely related to that of the goal tree (e.g., "+" for "-"). "Closely related" means, for our purposes, that there is a transformation axiom which can turn one operator into the other (i.e., the root of the axiom's LHS is the first operator, the root of the RHS is the closely related operator).

   (c) We order the selected M-ops so that those that most closely match the remainder of the tree come first. Specifically, in the case of a "←", those M-ops with the correct destination are put first. When the primary operator node is an operand class, which can represent several kinds of locations or constants, special checks are necessary to eliminate/order the possibilities.

(2) (TRANSFORM) For each of the feasible M-ops for the goal, in the above order, we attempt to transform the goal to match the M-op. This is done by matching the trees, node by node:

   (a) When a <u>match</u> occurs, we recurse on the descendents; if they match, we return successfully.

   (b) When a <u>mismatch</u> occurs, we select transformation axioms whose LHS and RHS match the root operators of the (sub-)goal and (sub-)M-op, respectively, and

apply them to transform the goal tree into new goal trees (this corresponds to the selection of operator by difference in means-ends analysis). For each new goal thus formed, we recursively attempt to transform it into the (sub-)M-op.

Note that the algorithm does not stop upon finding one code sequence for the given goal tree. It tries to transform the goal for each feasible instruction; each can lead to code sequences. Also, it tries all axioms applicable to a given mismatch when a mismatch occurs. All the possibilities for a node and its subnodes are then returned.[19] At the top level, the best code sequence found will be chosen according to a best-cost criterion, which is provided by the user as a function of time and space.

As an example of the use of transformations, consider the problem of loading the accumulator on the PDP-8 (there is no load instruction to do this directly). This can most easily be understood by following the steps of the actual search algorithms; a trace output from the implementation is shown below. The MOP description has been input at this point, and the top-level search routine is given the goal tree "(← %ACC %MP)", the TCOL representation of the problem of interest (the comments in italics have been inserted to annotate the output; also, parts have been truncated with ".." for readability; see Appendix D for the complete version):[20]

```
Search: (← %ACC %MP)                          *SEARCH is passed goal tree
Attempting M-op-match                         *no instructions match goal
Attempting Decompositions                     *(Decomp's are explained next section)
Attempting Transformations
 Feasible[1]: (← %ACC (+ %ACC $1:Z))          *attempt using TAD for goal
  Transform: (← %ACC %MP) => (← %ACC (+ %ACC $1:Z))
   Transform: %ACC => %ACC                    *LHS of "←" matches
```

---

[19] Actually keeping an enumeration of all possible code sequences and forming cross-products is not necessary. Instead a "disjunction" node is inserted into the result code (the result code is represented as a tree, except the only legal nodes are sequencing (;), disjunction (|), code (EMIT), and special pseudo-ops (LABEL, ALLOC)).

[20] The parenthesized LISP-like form used for the trees is explained in Appendix A. For example, (← %ACC (+ %ACC %MP)) means add a memory location (%MP) to the accumulator (%ACC). Parameters, e.g., "$1", are associated with nodes for reference, as explained in the appendix. Global paramters, e.g. "$$1", are parameters whose scope is over an entire search, as opposed to a single axiom or M-op; they are used to refer to temporaries needed in the code sequence, for example (see appendix). Access modes are preceded with "%" by convention. Operand classes (e.g., Z in the example) are not. The full text of all the examples in this chapter can be found in Appendix D, if the reader is interested in more details of any particular search.

```
 Transform: %MP => (+ %ACC $1:Z)                    *but RHS mismatches
  Applying $1 :: (+ 0 $1)  to: %MP                  *try axiom to reduce difference
   Transform: (+ 0 %MP) => (+ %ACC $1:Z)            *now "+" node matches
    Transform: 0 => %ACC                            *but 0 mismatches %ACC
     Applying Fetch Decomposition to: 0             *Acc←0 will fix this mismatch
      Search: (← %ACC 0)                            *and there is a CLRA M-op
      Attempting M-op-match                         *(M-op match explained later)
      M-op Match: (; (ALLOC $$2:Z) (EMIT[DCA 1 1 1] 3 $$2:Z)) *ignore this line for now
      M-op Match: (EMIT[CLRA 3 1 1] 7 0 20)
    Transform: %MP => $1:Z                          *Z is an OC which matches %Mp
Feasible[2]: (← %ACC (+ %ACC 1))                    *try other feasible M-ops...
 Transform: (← %ACC %MP) => (← %ACC (+ %ACC 1))
  ...                                               *but no other solutions found
Best Sequence is:
      [Alloc $$1:%ACC]
      CLRA
      TAD     %MP
-----------------------
```

The first feasible instruction found is the two's complement add (TAD) instruction, whose tree representation is "(← %ACC (+ %ACC $1:Z))"; no other instruction more closely matches the primary operator and also has the appropriate destination (%ACC). We therefore attempt to transform

```
(← %ACC %MP)   =>   (← %ACC (+ %ACC $1:Z)).
```

The %ACC part matches, but the RHSs mismatch. The program finds the transformation, $1 => (+ 0 $1), whose root operators match the mismatching subtrees, and it is applied to create the subproblem of transforming

```
(+ 0 %Mp)   =>   (+ %ACC $1:Z).
```

The "+"s now match, but the 0 and %ACC mismatch. Fetch decompostition is applied to make these match, by storing 0 into %ACC. Two instructions are found to do this (more on this later), the better one being CLRA (clear accumulator). The %MP matches the operand class Z, because Z is defined to allow either a direct or indirect memory reference on the PDP-8. We have then completed the match. The search proceeds to try other feasible instructions, but no further code sequences are found. The best code sequence to load %ACC is therefore to clear %ACC and add %MP.

As a second example of transformations, we will consider subtraction on the PDP-8. This is

probably one of the most difficult cases the code generator must handle: not only is there no subtract instruction on the PDP-8, there is not even a negate instruction.[21]

This example will illustrate the learning behavior exhibited by the algorithm. When a code sequence is found for a given goal tree, the goal-tree/code-sequence pair is stored away for later use. This is done in the implementation by storing the goal tree as if it were an M-op: if the goal tree is encountered again later, this Pseudo-M-op will be found, and the (multiple-instruction) code sequence previously found will be generated. Pseudo-M-ops will be discussed further in section 4.3.4. In this particular example, code sequences have previously been derived for loading the accumulator (the previous example), and for negating the accumulator (see appendix), as noted in the commentary below. In the use of the search algorithm, as will be described in section 4.4.1, the code generation cases are attempted in order from simpler to more complex cases, to maximize the re-use of earlier results. Of course, at the expense of greater search depth, this is not necessary.

```
Search: (← %ACC (- %MP %ACC))                    *goal is Acc←Mp-Acc
Attempting M-op-match
Attempting Decompositions
Attempting Transformations
 Feasible[1]: (← %ACC (- %ACC 1))
 *the above M-op (DECA) is chosen because its primary operator (-)
 *matches that of the goal, but it is useless for subtraction...
  Transform: (← %ACC (- %MP %ACC)) => (← %ACC (- %ACC 1))
   Transform: (- %MP %ACC) => (- %ACC 1)
    ...
 [fail on (← %ACC (- %MP %ACC)) ]
 Feasible[3]: (← %ACC (+ %ACC $1:Z))             *finally TAD is chosen as a possibility
  Transform: (← %ACC (- %MP %ACC)) => (← %ACC (+ %ACC $1:Z))
   Transform: %ACC => %ACC                       *destinations match, but
   Transform: (- %MP %ACC) => (+ %ACC $1:Z)      *source expressions mismatch
    *the following axiom is applied to attempt to correct the +/- mismatch
    Applying (- $1 $2) :: (+ $1 (- $2))  to: (- %MP %ACC)
     Transform: (+ %MP (- %ACC)) => (+ %ACC $1:Z)
      Transform: %MP => %ACC
       Applying Fetch Decomposition to: %MP using: $$4:%ACC
        Search: (← $$4:%ACC %MP)
        Attempting M-op-match
```

---

[21] The fact that there is only one accumulator also complicates matters, but this is a register allocation problem and does not concern the TRANSFORM algorithm (see section 3.2.3).

```
      M-op Match: (; (ALLOC $$5:%ACC) (EMIT[CLRA 3 1 1] 7 0 20) (EMIT[TAD 1 1 1] 1 %MP)
    Transform: (- %ACC) => $1:Z
     Applying Fetch Decomposition to: (- %ACC) using: $$6:%MP
      Search: (← $$6:%MP (- %ACC))              *$$6:%MP was allocated for $1:Z here
      Attempting M-op-match
      ...                                       *no soln is found using this approach,
    [fail on (- %ACC) ]                         *so try another axiom, commutativity(**):
    Applying (+ $1 $2) :: (+ $2 $1)  to: (+ %MP (- %ACC))
     Transform: (+ (- %ACC) %MP) => (+ %ACC $1:Z)
      Transform: (- %ACC) => %ACC
       Applying Fetch Decomposition to: (- %ACC) using: $$8:%ACC
        Search: (← $$8:%ACC (- %ACC))           *this quickly leads to a soln because
        Attempting M-op-match                   *M-op from earlier search found(**)
        M-op Match: (; (ALLOC $$9:%ACC) (EMIT[COMA 3 1 1] 7 0 40) (EMIT[INCA 3 1 1] ..))
      Transform: %MP => $1:Z                    *and RHSs match
  Feasible[4]: (← %ACC (+ %ACC 1))              *other M-ops are tried...
   Transform: (← %ACC (- %MP %ACC)) => (← %ACC (+ %ACC 1))
    ...                                         *but no other solutions found
Best Sequence is:
      [Alloc %ACC]
      COMA
      INCA
      TAD      %MP
-----------------------
```

Note the lines marked by (**), where it was necessary to use commutativity and the previously derived code sequence for negating the accumulator (COMA; INCA) to find a code sequence for the goal.

The search algorithm was also tried on other machines. An interesting example is computing an AND of two locations on the PDP-11; we will use this as the third and final example of transformations. There is no AND instruction, so it is necessary to apply axioms relating the AND, OR, and NOT operations:

```
 Search: (← $$1:%M (AND $$2:%M $$3:%M))         *goal is $$1:%M←$$2:%M AND $$3:%M
 Attempting M-op-match                          *the "$$n"s in a goal tree are distinguished
 Attempting Decompositions                      *from "$n"s in axioms/M-ops; see App. A
 Attempting Transformations                     BIC instruction is tried for the goal:
  Feasible[1]: (← $1:DST (AND $1:DST (NOT $2:SRC)))
   Transform: (← $$1:%M (AND $$2:%M $$3:%M)) => (← $1:DST (AND $1:DST (NOT $2:SRC)))
    Transform: $$1:%M => $1:DST                 *$$1:%M matches $1:DST, but
    Transform: (AND $$2:%M $$3:%M) => (AND $1:DST (NOT $2:SRC))
```

```
      Transform: $$2:%M => $1:DST                    *$$2:%M mismatches $1:DST, because (:*)
       Applying Fetch Decomposition to: $$2:%M using: $$1:%M
        Search: (← $$1:%M $$2:%M)                    *both occurences of $1:DST must be same
        Attempting M-op-match                        *so a move is generated (using FetchD)
        M-op Match: (EMIT[MOV 2 1 1] 1 $$2:%M $$1:%M)
      Transform: $$3:%M => (NOT $2:SRC)              *now only BIC node mismatching is NOT
       Applying $1 :: (NOT (NOT $1)) to: $$3:%M      *axiom invoked to resolve this
        Transform: (NOT (NOT $$3:%M)) => (NOT $2:SRC)
         Transform: (NOT $$3:%M) => $2:SRC           *must apply Fetch D, which causes
          Applying Fetch Decomposition to: (NOT $$3:%M) using: $$4:%R
           Search: (← $$4:%R (NOT $$3:%M))           *reg ($$4:%R) to be alloc'd for $2:SRC
          Attempting M-op-match
          Attempting Decompositions
          Attempting Transformations
           Feasible[1]: (← $1:DST (NOT $1:DST))      *try COMplement for the NOT
            Transform: (← $$4:%R (NOT $$3:%M)) => (← $1:DST (NOT $1:DST))
             Transform: $$4:%R => $1:DST             *destination matches, but again,
             Transform: (NOT $$3:%M) => (NOT $1:DST)
              Transform: $$3:%M => $1:DST            *both occurences of $1:DST must be same
               Applying Fetch Decomposition to: $$3:%M using: $$4:%R
                Search: (← $$4:%R $$3:%M)            *so another move is generated
                Attempting M-op-match                *this completes code seq using BIC
                M-op Match: (EMIT[MOV 2 1 1] 1 $$3:%M $$4:%R)
           Feasible[2]: (← $1:DST $2:SRC)
           ...
    Feasible[2]: (← $1:DST (NOT $1:DST))             *this M-op doesn't lead to code seq,
     Transform: (← $$1:%M (AND $$2:%M $$3:%M)) => (← $1:DST (NOT $1:DST))
      Transform: $$1:%M => $1:DST                    *although it could, by using
      Transform: (AND $$2:%M $$3:%M) => (NOT $1:DST) *DeMorgan's Law:
       Applying (AND $1 $2) :: (NOT (OR (NOT $1) (NOT $2))) to: (AND $$2:%M $$3:%M)
       ... *the search doesn't go deep enough to find this more roundabout sequence
Best Sequence is:
        [Alloc $$1:%M]
        MOV     $$2:%M $$1:%M
        [Alloc $$4:%R]
        MOV     $$3:%M $$4:%R
        COM     $$4:%R
        BIC     $$4:%R $$1:%M
-----------------------
```

The best code sequence therefore requires complementing one of the arguments of the AND,

then using the BIC instruction to AND the complement of the (now complemented) argument with the other argument. Also note the line marked with (**), which illustrates how parameters constrain the arguments of instructions. In the BIC instruction, for example, one of the source arguments, $1:DST, must be the same as the destination:

```
(← $1:DST (AND $1:DST (NOT $2:SRC)))
```

Since the locations specified by the goal tree, $$1:%M and $$2:%M, are not the same, Fetch Decomposition is applied to move $$2:%M to $$1:%M.

4.3.3 Decompositions

The second method applied in the search for optimal code sequences is decomposition by heuristic search. In this case, we have only a goal tree (as opposed to a goal tree plus feasible instructions in the last section):

(1) If the goal tree matches an instruction (or instructions), then the instruction(s) are returned as the code sequence(s) for the goal ("M-op Match"). Otherwise:

(2) For each decomposition axiom whose LHS matches the goal tree, we apply the axiom to create a new goal tree.

(3) For each new goal tree, we recursively search for code sequences. All resulting code sequences are returned.

As an example illustrating the use of decompositions, we will use the PDP-8 again, generating code for "If Acc=0 then Acc←1":

```
Search: (IF (EQL %ACC 0) (← %ACC 1))
Attempting M-op-match
Attempting Decompositions                    *first SEARCH tries applying defn of IF
Applying (IF $1 $2) :: (; (-> (NOT $1) $3:%MP) $2 (LABEL $3:%MP))
*note (-> A B) means "if A then goto B". LABEL means emit label.
 Simplifying (NOT (EQL %ACC 0)) to (NEQ %ACC 0)
 Search: (; (-> (NEQ %ACC 0) $$1:%MP) (← %ACC 1) (LABEL $$1:%MP))
 Attempting M-op-match
 Attempting Decompositions                   *SEARCH decomp's ";" node from IF defn
 Applying Sequence-Decomposition             *and treats each subnode as a subgoal
  Search: (-> (NEQ %ACC 0) $$1:%MP)          *1st subgoal (from IF-defn)
  Attempting M-op-match
```

```
  Attempting Decompositions
  Applying Skip-Decomposition                    *decomp into skip and goto
   Search: (GOTO $$1:%MP)                        *the goto
   Attempting M-op-match
   Attempting Decompositions
   Applying  (GOTO $1) :: (← %PC $1)
    Search: (← %PC $$1:%MP)
    Attempting M-op-match
    M-op Match: (EMIT[JMP 2 1 1] 5 $$1:%MP)
   Attempting Transformations
   Simplifying (NOT (NEQ %ACC 0)) to (EQL %ACC 0)
   Search: (-> (EQL %ACC 0) (+ %PC 1))           *the skip
   Attempting M-op-match
   M-op Match: (EMIT[SKPE 3 1 1] 7 1 5)
     ...
  Breadth Limit Reached(6)
  Search: (← %ACC 1)                             *2nd subgoal (of IF-defn)
  Attempting M-op-match
  M-op Match: (EMIT[SET1A.3 1 1] 7 0 30)
  Search: (LABEL $$1:%MP)                        *last subgoal (of IF-defn)
*now have "SKPE; JMP $$1; SET1A; $$1:" sequence
*SEARCH tries applying Skip-decomp to original goal instead (of IF-defn)
 Applying Skip-Decomposition
  Search: (← %ACC 1)                             *first try THEN-part
  Attempting M-op-match
  M-op Match: (EMIT[SET1A 3 1 1] 7 0 30)         *OK, it takes 1 instr
  Simplifying (NOT (EQL %ACC 0)) to (NEQ %ACC 0)
  Search: (-> (NEQ %ACC 0) (+ %PC 1))            *now try cond skip of 1
  Attempting M-op-match
  M-op Match: (EMIT[SKPNE 3 1 1] 7 1 2)          *success. now have 2 seqs.
 Attempting Transformations                      *keeps trying, but no more solutions..
   ...
-----------------------
Nodes Examined: 15
Est. Seconds:  .620@-1
Result Sequence(s):
<<<<<<<<<<<<<<<<<<<<<<<<
       SKPE
       JMP      $$1:%MP
       SET1A
$$1:
-----------or-----------
```

```
      SKPNE
      SET1A
>>>>>>>>>>>>>>>>>>>>>>>>
Best Sequence is:
      SKPNE
      SET1A
------------------------
```

Both the definition of IF and skip-decomposition get applied in this derivation, and two alternative code sequences are found depending on which is tried. The better sequence is to do a SKPNE (skip if accumulator non-zero) followed by SET1A (set accumulator to 1).

Note that in the search for code sequences, decompositions are applied to the goal first. This normally decomposes it into a sequence of more primitive control constructs, such as conditional and unconditional jumps. These constructs then either match M-ops directly, as they did above, or the TRANFORM algorithm may take over. On a machine with "condition codes" such as the Motorola 6800 or the PDP-11, a conditional jump may require several transformations. The TRANSFORM algorithm must handle relational operators on these machines by using Fetch Decomposition on boolean results in condition codes. For example, on the PDP-11, consider the case of an inequality test in a flow context (conditional jump):

```
Search: (-> (NEQ $$1:%M $$2:%M) $$3)          *jump to $$3 if %M locations not equal
Attempting M-op-match
Attempting Decompositions
 ...                                          *decompositions don't help
Attempting Transformations
 Feasible[1]: (← $1:DST (NOT $1:DST))
 Feasible[2]: (-> (NOT %Z) $1:ADR)            *attempt to transform into BNE instr
  Transform: (-> (NEQ $$1:%M $$2:%M) $$3) => (-> (NOT %Z) $1:ADR)
   Transform: (NEQ $$1:%M $$2:%M) => (NOT %Z)  *must apply defn of NEQ using EQL
    Applying (NEQ $1 $2) :: (NOT (EQL $1 $2))  to: (NEQ $$1:%M $$2:%M)
     Transform: (NOT (EQL $$1:%M $$2:%M)) => (NOT %Z)
      Transform: (EQL $$1:%M $$2:%M) => %Z     *now is the key Fetch D:
       Applying Fetch Decomposition to: (EQL $$1:%M $$2:%M) using: $$3:%Z
        Search: (← $$3:%Z (EQL $$1:%M $$2:%M))
        Attempting M-op-match                  *and we find a match: we're done
        M-op Match: (EMIT[CMP 2 1 1] 2 $$2:%M $$1:%M)
   Transform: $$3 => $1:ADR                    *user's label is $$3
 Feasible[3]: (-> (NOT %N) $1:ADR)             *continue looking for other seqs..
 ...                                           *but no more found
Best Sequence is:
```

```
      CMP     $$2:ZM $$1:ZM
      BNE     $$3
-----------------------
```

After considering several examples of transformations and decompositions, the essence of both the SEARCH and TRANSFORM algorithms should now be clearer. It may be useful to contrast them at this point. Note, for example, that TRANSFORM guides the application of axioms by using a feasible instruction selected for the goal, while SEARCH [almost] blindly applies axioms. TRANSFORM is thus a more powerful method; however, the means-ends analysis does not in general succeed with control constructs, and so we fall back on SEARCH in these cases (in actuality, both SEARCH and TRANSFORM are tried at the statement level; if the M-ops are high-level enough or goal tree low-level enough for TRANSFORM to succeed, the best code sequence is of course chosen). The relatively small search space makes this feasible: for example, there were only two ways to implement the IF in the first example in this section.

The reader may have noted that decomposition axioms are only applied at the top (root) node of the goal tree; SEARCH does not apply axioms elsewhere in the tree (unless an axiom applied at the top requires the subnode as a subgoal). Axioms need only be applied at the top node because decompositions are being used to decompose control constructs into sequences of more primitive constructs. It would be possible to fall back on applying axioms blindly to all points of the goal tree rather than the top node, but it was not necessary for the actual code generation cases encountered, so this was not attempted. This strategy would probably be computationally infeasible except for limited cases, anyway.

4.3.4 Compensations

The third procedure used in the derivation of code generators is a pre-pass on the MOP, performed by the INDEX algorithm. For each M-op, the following steps are performed:

(1) To allow efficient selection of M-ops (either to find feasible instructions in TRANSFORM or to check for M-op match in SEARCH), the M-ops are indexed according to their primary operators. This is simply a heuristic to speed the lookup of M-ops later.

(2) If the root operator of an M-op is ";", meaning that it involves several actions[22],

---

[22] We are using ";" in its sense in ISP for purposes of the M-op trees, not as a sequencing operation. That is, "A; B; C" means A, B, and C may be performed in any order and have no interaction. This independence of the M-op input/output assertions makes possible this algorithm for dealing with multiple actions.

special checks are made to determine if it might be usable for any of its several effects separately. We index each sub-action of the M-op as follows:

(a) If any co-action (other sub-action besides the current one) has a side-effect on a Reserved-type storage base, then this sub-action is not indexed.

(b) Otherwise, we create a new *Pseudo-M-op* whose LHS is the sub-action and whose RHS is the M-op RHS except with the addition of a *co-action list*, which, when the Pseudo-M-op is used, causes the insertion of the appropriate compensations for the side-effects. For example, if a side effect is on a General-purpose-type storage base, a storage-allocation pseudo-operation, ALLOC, would be inserted for the SB. If a side-effect is an increment of the PC, a No-op instruction is inserted as the compensation.

These Pseudo-M-ops are treated as M-ops during the search phase (SEARCH, TRANSFORM), when looking for an M-op matching a given goal tree. This lookup is somewhat complex. First, a set of possible M-ops is selected for a goal tree using the primary-operator hash scheme above. Then, to determine if each of these M-ops (or Pseudo-M-ops) matches the given tree, four cases must be considered:

(1) Both M-op and goal are single actions: in this case, we simply test whether the trees match.

(2) M-op is a multiple action, goal is a single action: match goal against sub-action, insert compensations for other sub-actions.

(3) M-op is a single action, goal is a multiple action: no match. (Goal will be decomposed into its sub-actions by the *sequence-decomposition* rule, at which point the M-op may match one of the sub-actions.)

(4) Both M-op and goal are multiple actions: test that every sub-action of the goal matches a sub-action of the M-op, then insert compensations for unused M-op sub-actions.

Dealing with side effects is thus done in two parts: indexing the M-ops under the sub-actions they perform, and then applying the compensation axioms to construct the code when the M-ops are retrieved during the search for code sequences. This two part algorithm avoids dealing with side effects as part of the search itself.

Incidentally, the four steps above could be performed at code generation time also. A slight variation of step (4) is being considered in the PQCC code generator to improve code quality: At a ";" node, as many as possible of the sub-actions are subsumed at each step. The source sequence "...I←I+1; If I=0 then..." might be subsumed by an "Increment-and-Skip-if-Zero" instruction. (These optimizations could alternatively be detected in peephole optimization.)

As an example, consider the deposit-and-clear-accumulator instruction on the PDP-8, DCA. It can be used for either of its two sub-actions ((1) depositing the accumulator in a memory location, and (2) clearing the accumulator) by inserting an ALLOC for the other location effected:

```
 Search: (← %ACC 0)
 Attempting M-op-match
   ...
Result Sequence(s):
<<<<<<<<<<<<<<<<<<<<<<<<
       [Alloc $$1:%MP]
       DCA     $$1:%MP
----------or----------
       CLRA
>>>>>>>>>>>>>>>>>>>>>>>>
Best Sequence is:
       CLRA
------------------------


 Search: (← %MP %ACC)
 ...
Best Sequence is:
       [Alloc %ACC]
       DCA     %MP
------------------------
```

In the first case, we use DCA to clear %ACC by allocating a memory location ($$1:%MP) into which %ACC can be stored (the search also finds the CLRA instruction which clears %ACC directly at lower cost in this case). In the second case, we use DCA to store %ACC in a memory location; but the compensation axioms have inserted the warning that %ACC is destroyed in this process, as shown by the [Alloc %ACC] in the output code.

A more common example of multiple-action instructions are the arithmetic operations on machines with condition codes. These instructions can be used for the primary arithmetic

operation performed, ignoring the effects on the condition codes, which are Temporary-type storage bases. All of the arithmetic instructions on the PDP-11 are of this form.

Side effects on the program counter are also handled by compensations. Increment-and-skip-if-zero (ISZ) can be used in conjunction with a no-op to get just the effect of the increment:

```
 Search: (← %MP (+ %MP 1))
  ...
Best Sequence is:
      ISZ     %MP
      NOP
-----------------------
```

4.3.5 Limiting the Search

There are parameters to control the extent of the search. Without these, the search could go on forever. The cutoff criteria are:

(1) A maximum depth of search. The depth of search is increased by one for each recursive application of a decomposition (in SEARCH) *or* transformation (in TRANSFORM).

(2) A minimum and maximum breadth. These are used in conjunction with FINDFEASIBLES, and are in units of search cost (the number of nodes in the search tree). We continue to try feasible instructions until the minimum number of nodes have been searched and a solution has been found, or until the maximum number of nodes have been searched and *no* solution has been found.[23]

A nice property of the cutoff parameters is that we can trade off between the speed with which a solution is found and the quality of the resulting solution (in terms of optimality). We could try searching for a long time if we are interested in optimal code. Also, we can automatically increase the parameters if no solution at all is found.

---

[23] For better performance, these limits are decreased with search depth, but this is not important to an understanding of their purpose.

Another nice property of the search is that it is possible to vary the code cost function supplied to the search routines to generate code optimized for space, time, or any ratio thereof.

The axioms could also be changed, to modify the assumptions the code generator makes; for example, whether floating point multiplies can be computed in any order.

4.3.6 The Search Space

The reader may be curious about the quantitative properties of the search space defined for this problem. Although the primary contribution in this chapter is providing a reasonably general solution to the machine-independent code generation problem at all, rather than in achieving some new level of performance, some rough data may be useful to give some insight into the nature of the problem and its solution.

The goal tree (the desired action) is the starting point in the search space. Each application of an axiom leads to a new node in the space (a new goal); this is true for both SEARCH and TRANSFORM. For the axiom set used, the branching factor, i.e., the number of axioms (or feasible instructions) attempted at each point, is typically about 2.5; the depth of the search tree varies widely according to the goal tree. For a typical machine, in fact, the vast majority of the templates for which code sequences are required are satisfied immediately by machine instructions (depth=1). But for the "interesting" problems that make the search necessary, the depth typically ranges 3 to 7 to find the best code sequence, leading to a typical search space size of a few dozen nodes.

The reader familiar with search problems such a computer chess will recognize this as a relatively small search space. This is in fact a key factor in making the search practical. Although the use of multiple methods reduced the size of the search space, the problem domain itself, *when suitably represented*, is not unmanagably large. The choice of representation is important in several dimensions of this work; for example, the representation of instructions and addressing, the use of trees as a common notation for matching, and the use of axioms to represent the legal moves in the search space. These representations lead to relatively straightforward algorithms.

4.3.7 Completeness and Optimality

Note that the search algorithm presented in the previous sections does not guarantee optimal code, or any code at all for that matter, because the search may not be deep enough to discover the equivalence. Furthermore, even if we searched to an arbitrary depth, a code

sequence might still not be found, because a necessary axiom to determine the code sequence's equivalence to the goal tree may not be in the axiom repertoire. This is not an error in the construction of the axioms. Unsolvablility of program equivalence is based on the fact that no set of axioms can express all the equivalences that are true over program trees.

This indicates that no one will ever be able to construct a program which satisfies the ultimate goal of this work: to take an arbitrary machine description and generate code. Fortunately, as in the field of proving programs correct, such theoretical results do not have great practical impact. For "real" machines (and "real" programs) a relatively small set of axioms seems to be adequate.

4.4. Code Generator Generation

4.4.1 Case Selection

We have now discussed the three main algorithms used in finding code sequences. The reader may want to refer to them in Figure 11 at this point. INDEX is applied as a pre-pass. SEARCH is the central search routine: it tests for termination (i.e. a M-op match), tries applying decompositions, and calls TRANSFORM to try transformations. We will now discuss how these routines are used with the final routine in the figure, SELECT, in the generation of code generators.

Considered as a whole, SEARCH, TRANSFORM, and INDEX constitute a machine-independent code generator. That is, SEARCH generates code for TCOL trees, for any given machine. In theory, this code generator could be used directly in a compiler. In practice, however, this would probably be too slow for general use: although considerable speedup of this specific implementation could be achieved since little attention was paid to the efficiency of this prototype, it is unlikely that we could do much better than 1000 instructions per second. While generating code machine-independently is useful in itself, it is of greater practical impact if comparable with conventional compilers in speed as well.

The alternative that comes to mind is to tailor the code generator to the machine, so that:

(1) The algorithm is much simpler, not requiring all the axioms about program equivalence.

(2) The algorithm can go directly to the best solution, alternatives having already been explored and rejected implicitly.

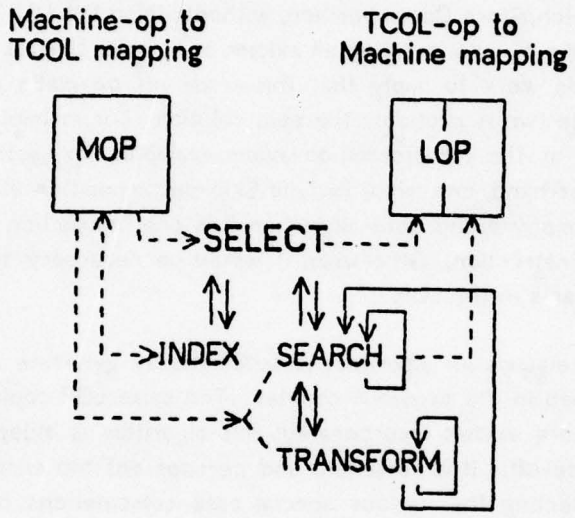


Figure 11: The flow of control (solid lines) and data (dashed lines) in the code generator generator. All routines use the MOP table; INDEX modifies it. The templates in the LOP table are derived by SELECT, which selects the special-case tree patterns, and SEARCH, which determines the code sequences for these tree patterns.

This desire to separate compile-time and compiler-compile-time is the reason for the existence of the LOP and the LOP-driven code generation scheme described in the previous chapter. The only axiom incorporated in the basic code generation algorithm discussed there is Fetch/Store Decomposition, which is performed by subtargetting to allow composition of the LOP patterns. Of course, the code generator generator, which constructs the LOP, has made use of many axioms.

There is a continuum of alternatives in the trade-off between the number of axioms incorporated in the code generator as opposed to the code generator generator. This ranges from the minimum of Fetch/Store Decomposition, without which the LOP patterns could not be composed, to the extreme of incorporating all axioms at compile time as mentioned earlier. It is not the intent of this work to imply that this trade-off be made at either extreme. A compromise between the two is probably the best solution. For example, there is no need to perform searches such as the transformation-axiom examples (in section 4.3.2) in the code generator. On the other hand, one might include Skip-decomposition at compile-time, so that IF B THEN S could be implemented optimally when S is one instruction long and the machine has a conditional skip instruction. Otherwise, it would be necessary to include all cases of one-instruction THEN-parts in the LOP.

In this section we will discuss an algorithm to automatically generate a LOP for the minimal code generator described in the previous chapter. The same LOP could also be used with a code generator with more axioms incorporated: this algorithm is independent of the choice made in the above trade-off. It is fortunate and perhaps not too surprising that this is so, because otherwise detecting the various special case combinations of language operators would have made compilers unreasonably complex.

The basic purpose of this (SELECT) algorithm is to insure there is a template in the LOP for every basic arithmetic computation, conditional, and control construct. The algorithm derives the entries in the LOP in five steps:

(1) Include all the M-ops in the LOP. Figuratively, set LOP←MOP. Thus, if a program tree segment matches an M-op directly, the M-op will be selected to code the tree.

(2) To these, add the Pseudo-M-ops defined by INDEX, which allow the use of the M-ops in additional cases (namely, for their partial effects).

(3) Insure there is a template for A←B, for every pair of distinct access modes A and B such that A and B are "simple" references to locations of the same size. A "simple" reference is one in which the index into the storage base is a constant or open constant (as opposed to, say, indirect or relative addressing). If there is already such an entry from steps (1) and (2), no action is taken. Otherwise, SEARCH is called to find

the best code sequence for A←B, and if a solution is found (there may be no way to do this move), a template is created whose pattern (LHS) is the A←B tree and whose result sequence (RHS) is the code sequence.

(4) Insure there are templates in the LOP for every operator in value and flow contexts. This is done similarly to the previous step, calling SEARCH for every tree of the form "A←B op C", "A←op B", and "A op B -> C". A, B, and C may be any access modes, since the code generator will make any required moves to perform operation "op" on data in other locations. For example, if logical "AND" didn't exist as the primary operator of an M-op directly on the machine, a code sequence for it would be derived (by calling SEARCH) and the resulting template (LHS is the AND tree, RHS is the derived code sequence) added to the LOP. All derived templates are also indexed as Pseudo-M-ops as discussed in section 4.3.2, for use in further searches.

(5) Finally, add to the LOP the productions for control operators. These correspond to the axioms in Figure 10 which define WHILE-DO, IF-THEN-ELSE, etc., in terms of conditional and unconditional jumps. These templates are machine-independent, because conditional jumps have already been handled in step (4).

The reader may wish to refer to Figure 11 one more time to see the relationship between this top-level algorithm (SELECT) and the algorithms previously discussed (INDEX, SEARCH). The LOP is figuratively divided into two parts to show that SELECT chooses the LHSs of productions, and SEARCH chooses the RHSs. INDEX is shown augmenting the MOP table with the derived Pseudo-M-ops. In Appendix F, an example trace of the SELECT algorithm is given.

This algorithm insures that the minimal code generator using the LOP will be able to generate code for all TCOL operators, and that if there exists a one-instruction code sequence for a subtree, it will find it. It does <u>not</u> guarantee that if the search algorithm discussed in the previous section generates optimal code that the code generator using the LOP generated therefrom will do so, because the necessary special case combination of TCOL operators may not have been included in the LOP. Interestingly, however, this rarely happens with the simple SELECT algorithm suggested: special cases more complex than single TCOL operators (in all contexts) are not normally needed, except for those cases which match M-ops directly (which SELECT handles in step (4)).

It is intuitively unsatisfying, however, that this approximation does well for existing machines. We would like a scheme to automatically determine what special cases should go into the LOP to insure optimal code. Another solution would be to tailor the code generator proposed in the last section to make it efficient for each specific machine, for example by including only

the necessary axioms. The tailored code generator would then go into the compiler directly. Further exploration of these alternatives and other possible solutions has been left to future research.

4.4.2 Inter-State Optimization

Even if the axioms were complete, and we searched to arbitrary depth, the search algorithm does not guarantee optimal code. The algorithm does not simulate the processor state to keep track of the current contents of the various locations. Common sub-expressions are not recognized. If a value is required in a register the code generator will re-load it even if it already happened to be there. This isn't a problem with inadequate special case analysis due to an inadequate set of axioms; the code preceding the re-load may have been generated from an arbitrarily distant piece of the program tree.

This kind of optimization is probably hopeless to deal with in any local tree context analysis. However, this immediately suggests the solution to this problem: it is most efficiently handled outside of the code generator (see Wulf et al[1975]):

(1) before code generation, common sub-expression analysis has been performed, and

(2) after code generation, peephole optimizations collapse redundant operations.

A second problem also arises from the search algorithm's ignorance of the machine state: we must check that it has not inconsistently used storage. Namely, in applying Fetch/Store Decomposition, an unsatisfiable set of allocation commands may be emitted: for example, there may not be enough locations of the required storage base type. This problem can be corrected quite simply by doing a post-test on the solutions generated by the search algorithm. Specifically, we could put the best solution which does not violate allocation constraints in the LOP (this is the generate-and-test paradigm from artificial intelligence).

On the other hand, this problem suggests some alternative approaches:

(1) Do a simulation of the processor state in parallel with the search as described. This is complicated by the fact that we are finding multiple solutions, typically with different effects on the processor state, that locations may overlap in strange ways on the machine, and that we cannot guarantee that the symbolic simplifications will always detect equivalent expressions when they occur.

(2) It is possible to use an entirely different representation than trees in the search for

code sequences. In particular, an interesting alternative would be letting the nodes in the search space be states of the machine, described as a set of simultaneous equations over the processor state location values. M-ops transform one node in this space into another. In the starting state, all locations S have their initial values S*, and the goal state is to achieve a given set of equations, e.g., %ACC=%MP*+%ACC*. This approach has some disadvantages and advantages with respect to the approach taken here; exploring this alternative to see if the AI methods could still be used practically in this less structured representation would be a good topic for further research.

Further ideas in these areas may come from PQCC work on the last compiler phase (peephole optimization) or earlier compiler phases (common sub-expression analysis)

4.4.3 Using the LOP

Given that we have constructed the LOP, how do we use it in a compiler? This is the problem of the compiler construction phase of Figure 1.

The LOP table can be used in a variety of ways. Although in the previous discussions we've assumed the code generator is table-driven, it would also be possible to construct the actual source code for the code generator from the LOP table. However, entirely adequate speed and flexibility was obtained with the table-driven code generator implemented in Chapter 3; so the approach suggested here is a simple program, BUILD, which translates the LOP into tables that can be compiled by BLISS and loaded with the compiler source code. Facilities in the BLISS language (Wulf et al [1970]) greatly simplify the construction of the tables of tree patterns and result sequences at compile time[24], but tables could of course be generated for any language.

Because the LOP is human readable/writable (before translation into program tables), it is possible to manually modify or augment the table (the syntax of the LOP is similar to that of the MOP, see appendix E). Programs can also be written to read, modify, and rewrite the LOP table; this might be desirable to extract special information, or to optimize it in some way. However, if the LOP is manually modified, it would be desirable to verify that the modifications maintain the correctness of the table, by proving that the LHS and RHS of the modified or added productions are semantically equivalent under the TCOL equivalence axioms. This is beyond the scope of this thesis, but previous work in this area has been successful on this kind of task (Samet[1975]).

---

[24] particularly the pre-loaded data ("plit") and source-file inclusion ("require") constructs

The LOP is not just used in the code generation phase of the compiler. The template productions, for example, are required in register allocation and other phases of an optimizing compiler, as described in section 3.7. From a human-engineering point of view, it would be desirable to centralize the machine-dependent data base of tables used by the compiler, in human-readable tables. The LOP could be used for this function, although some extensions and modifications to the format as suggested in this thesis would probably be desirable.

It should be noted that the compiler we have been discussing is a *cross*-compiler, i.e., the machine-dependent tables are to be used by source code compiled to run on the PDP-10. However, a likely possibility for future work would be to bootstrap the compiler with itself, i.e., compile itself into code for the target machine. A translator from BLISS to TCOL is relatively straightforward; the main aspect which would require special attention to allow the bootstrapping would be dealing with word size differences and memory size limits

4.5. Relation to Other Work

There has been very little work prior to this one in the area of automatic derivation of code generators, and even less successful work. In Cattell[1977], there are discussions of most of this work; however, the most closely related work will be mentioned here.

Newcomer[1975] was a predecessor to this work, and contributed in several ways. The principal ideas shared with Newcomer's work are the use of means-ends analysis to find code sequences, and the use of tree templates as the central representation. This work *differs* in two main respects. The most important one is probably the machine model. While Newcomer used trees to represent instructions, the essential semantics with respect to the means-ends analysis were encoded in "attributes" which must be set up by the user of the system. Newcomer suggests attributes, for example, that specify where the result of a tree expression is stored, or whether the tree has the correct or inverted sign. In this work, attributes have been dropped; we need only the semantics of the TCOL operators. (Incidentally, although attributes were dropped for code generator generation, they may well be the best general mechanism for dealing with optimizations in the DELAY phase of the compiler.) As should be apparent from Chapter 2, the model of the machine is more general, dealing with control constructs, side effects, binary representation, and so on. The *second* main difference in this work is the use of other methods besides means-ends analysis to deal with control constructs and side effects.

Another predecessor of this work is Samet[1975]. Samet's goal was not to generate code, but rather to verify that the code generated for a source language tree is in fact correct.

However, there are some similarities in that Samet also used an axiomatization of the equivalence of trees. His work also might be used in the verification of the LOP, as mentioned earlier.

The reader who is interested in other work in the area of this thesis should also see three theses that were completed quite recently (all less than six months at the time of this writing): Fraser[1977], Ripken[1977], and Granville[1977]. These were not completed at the time of the survey (Cattell[1977]), so a short comparison of approach may be helpful to the reader here.

Fraser designed a *human-knowledge-based* code generator taking an ISP description as input. His central algorithm consists of pattern matching common cases which the system "understands". For example, the program explicitly checks for machines with conditional skips as opposed to conditional-jump architectures. The observation that makes this approach possible is that most current computer architectures are quite similar in design, and consequently it is possible to base the system on a manageable number of cases (Fraser presents evidence that the amount of new programming knowledge that must be added decreases as new machines of similar architecture are added). In contrast, as should be apparent by now, the present work was to test the feasibility of taking a more formal approach, using equivalence axioms rather than built-in programming knowledge to minimize the machine-dependency of the system. The main *disadvantage* of the formal approach is that it is potentially combinatorially explosive, since it does not directly match "built-in" special cases; but this chapter has presented evidence that it can in fact be done practically. This does not mean there is no longer a need for the human-knowledge based approach. The best approach is probably a combination, using formal methods plus human-knowledge special cases to make decisions that are infeasible to otherwise automate for some reason.

The other two theses (Ripken and Glanville) are principally concerned with code generation rather than with code generator generation (at least as defined by Fraser and this work). They both assume a one-to-one correspondence between the machine and language operators. For example, they cannot generate code for the cases such as those given as examples in this chapter: operators which do not exist on the machine, e.g., AND on the PDP-11, or simply loading the accumulator on the PDP-8, as well as control constructs. It is therefore more appropriate to compare these works to Chapter 3 rather than Chapter 4 of this thesis.

Glanville's work is an extension of that of Weingart[1973], using the more recent LR(K) parser technology. This algorithm is similar to the MMM algorithm in chapter 3: program trees are matched against instructions represented in the form of a grammar.

Ripken's scheme is quite sophisticated, as his goal is generating near-optimal code. Ripken

deals with the interaction of code generation and temporary allocation in detail, using multiple passes on the program tree. It is interesting to constrast his scheme to the DELAY-TNBIND-CODE scheme; the models have arrived at similar conclusions with respect to the necessary structures to generate good code. Ripken did not implement his model, so it is hard to evaluate whether his dynamic-programming algorithm can be implemented practically, or whether he can properly deal with the details of real machines; however, it is clear that Ripken has studied the problem thoroughly, and an implementation should be forthcoming.[25]

4.6. Implementation

In Appendix F is a list of the axioms used in the search. The MOPs for the Mini-S and PDP-11, used for the examples in this chapter, can be found in Appendix B, along with an explanation of the syntax used for the representation. In Appendix D, traces of the search algorithm for various examples are given. In particular, the full text of the examples in this chapter can be found there. In Appendix E, a trace of the generation of the LOP is shown for the PDP-11.

---

[25] I'd like to thank Bert Speelpenning [1978] for his English summary of Ripken's work.

5. Results and Conclusions

"A conclusion is the place where you got tired thinking"
- Martin H. Fischer

5.1. Summary

This dissertation has presented: (1) a model of instruction set processors, (2) a code generation algorithm in which machine-dependent information is separated into tabular form, and (3) a scheme for heuristic search for optimal code sequences, based on an axiomatization of tree equivalence. Each of these ideas has been studied in some depth. The crucial representations and algorithms have been implemented to test the consistency of the ideas and to evaluate empirically the practicality of their use in the generation of code for real machines.

Evaluation of work in areas of this complexity is crucial. This is apparent to anyone who has tried to read previous work whose strengths and weaknesses are not discussed, requiring a painful analysis of the details in almost as much depth as the original work. On the other hand, first-hand evaluation of work is difficult, as the limitations are normally either not understood or are overlooked by the original author (otherwise, they would very likely not be limitations). With this in mind, let's try to examine at least those contributions and limitations that are apparent. The next section provides some quantitative and qualitative data on the results of the thesis. The remaining two sections of the chapter then summarize the contributions and limitations.

5.2. Results

The results have been encouraging. The machine representation appears to be general enough to deal with a variety of actual machine architectures, and the representation is extendable to deal manually or automatically with unusual features that do not directly fit the model. The code generation algorithm satisfies the goals of tabularizing machine dependence and at the same time remaining simple, flexible, and fast enough for use in a production compiler. The last and perhaps most interesting result is that the formal approach to heuristic search for code sequences was successful in finding optimal code sequences for real machines.

One might expect the code generator to be relatively slow, since it involves a table-driven

pattern-matching scheme. However, the prototype implementation on a PDP-10/KL10 is basically I/O bound, generating about 2000 instructions per second. The code itself is quite compact, requiring only 1K 36-bit words, because all the machine-dependent information is in the tables, which require more space (the amount being target-machine dependent, but order of 10K words). These figures can only be regarded as estimates until the code generator has been interfaced with the PQCC compiler under development. The complete compiler will also be necessary for an objective evaluation of the code quality, although the example in Chapter 3 provides a limited demonstration of the code optimality.

The code generator generator is also surprisingly fast in comparison to previous results in the area (Newcomer [1975]). The example derivations of code sequences in Chapter 4 typically took about .1 seconds (KL10). The generation of the LOP itself took about 10 seconds for the PDP-11, as shown in appendix E. The code generator generator uses 40K words plus 10 to 20K data; it is implemented in SAIL (Reiser et al [1976]).

The speed of the code generator generator is not greatly affected by either the number of axioms or the number of instructions on the target machine. This is because the primary operator indexing scheme allows the search routines to go almost directly to the applicable axiom (for a mismatch) or instruction (for a goal tree). Note also that the axioms are machine-independent, so that it should only be necessary to add new axioms when a new domain is added, e.g., when TCOL is extended to include a new data type such as character strings.

Probably the most impressive result, although difficult to quantify, is the scope of machine architectures the code generation scheme handles. To evaluate this, we will consider a cross-section of common architectures: the IBM 360, PDP-10, PDP-11, Intel 8080, Motorola 6800, and PDP-8. The proposed scheme is capable of generating a code generator for all of these machines with certain restrictions; the following discussion will therefore concentrate on the restrictions.

The heuristic search algorithm itself was quite successful in cases where the machine fit the model; the restrictions are primarily with respect to the machine model. With regard to these restrictions, there are four main areas which should be considered:

(1) The top-level definition of the instruction interpreter, which fetches instructions from memory and acts according to the input/output assertions. It is surprising how well this simple scheme fits so many architectures. However, certains instructions, such as the XCT instruction on the PDP-10 which recursively calls the instruction interpreter, and the "micro" instruction on the PDP-8 which can independently execute 9 simple actions, do not fit this scheme.

(2) Instructions with multiple actions. Previous work has not tried to deal with these. Some examples would be ISZ on the PDP-8, BCT on the IBM 360, and use of auto-increment on the PDP-11. This thesis has presented a scheme for dealing with these, when the program tree contains the matching actions in immediate succession, although rearrangements of the program tree to make these applicable are not considered (this would occur in DELAY in the PQCC model).

(3) Data types and axioms. It is necessary to extend TCOL to deal with special machine data types. The algorithm will not determine what they are intended for: it is necessary to have axioms describing their properties and relationships to program constructs. For example, the character and decimal arithmetic instructions on the IBM 360 fall in this class, as does byte manipulation and block transfer on the PDP-10. Also not covered by the axioms in the actual implementation are the properties of shifting and testing of bits within a word, and special arithmetic properties, namely carry and overflow (the code generator simply ignores overflow on the PDP-11 examples given).

(4) The representation of storage. The Operand Classes and Access Modes successfully deal with all these machines, including the PDP-11 and Motorola 6800 which have many addressing modes. Note that the code generator is not fooled by mnemonics as a person might be; for example, it will use indexing (intended for address computation) to do addition if it is optimal in the context of its use. Also, the Storage Base scheme can deal with machines with many register types (Intel 8080), one accumulator (PDP-8), and many different operand sizes (IBM 360). However, note that the problems concerned with the allocation of these registers are not part of this thesis; work is under way in PQCC on this problem.

5.3. Contributions

All three of the main chapters of this thesis contain potential contributions. Some ideas that might prove useful are:

(1) The model of machines, including the input/output assertion representation of instructions, the separation of the addressing functions, and the attention paid down to the bit-level representation of data and instructions. This model is not just for code generation purposes; it can be used for other applications, and also makes suggestions for machine description languages. The importance of the machine model is that it defines and restricts the class of objects that we are dealing with; this formalism is a key to making this work possible.

(2) The code generation algorithm, including the tabular representation of the process, and the ideas for interaction with other components of a compiler, such as register allocation. Previous work has not separated machine-dependence to this degree, or dealt with its interaction with the other phases of an optimizing compiler. The MMM algorithm is also a new idea: it could prove to be a good compromise between optimality and fast code generation. Note that the formalization of the code generation process is not only necessary for code generator generation, but is independently useful.

(3) The axioms for tree equivalence (in particular those concerned with programs and machines), the heuristic search algorithm, which includes the application of several methods, and, finally, the ideas for the use of this algorithm for the automatic generation of code generators. The search algorithm and axioms are probably the most significant contributions of the thesis. The central reason for the success of the work is largely the *representation*: that is, the algorithms are relatively straightforward once the problems have been represented. This can be seen to apply in several areas of the work, including the machine repesentation, code generator repesentation, and the use of axioms and trees in the search for code sequences.

Some of the techniques used in this work may be applicable to other applications of machine descriptions. For example, automated hardware generation is conceptually analogous to code generation, as it involves decomposing a given algorithm into a set of given primitives (Leive [1977]). As just mentioned, this thesis illustrates the principle that a problem is often easy once it has been precisely and flexibly represented. Some important representational issues in this work were:

(1) The use of a common notation, TCOL, to represent procedural semantics. Also important is the extensibility of TCOL with respect to new data types and operators.

(2) The restricted form of the instruction interpreter, reducing the selection of primitives to sequences of actions represented by input/output assertions.

(3) Abstraction of orthogonal properties such as addressing and binary representation from the representation of the abstract operations themselves (the instructions).

5.4. Future Work

A strong point of this thesis is that it ties together the rather disparate areas of machine representation, optimizing compilers, and heuristic search. Previous work has not attempted to bridge these gaps, so this broad scope was badly needed. However, as many new questions have arisen as have been resolved. The relatively broad scope of the thesis, although necessary to bridge the gaps in previous work, dictates that only some of the problems have been studied in detail. There is room for future research in all three areas of the thesis. This future work includes:

(1) Generalizations in the machine model, to deal with a wider range of architectures. Some extensions to the model that would be useful are:

   (a) A concise representation of data types, including a way to deal with phenomena such as arithmetic overflow and complex instructions such as character string manipulation.

   (b) A way to describe input/output.

   (c) A way to take into account special machine features such as multiple ALUs, instruction lookahead, pipelines, or caches in the code optimization (probably after code generation).

(2) Further research on compilers, namely:

   (a) Optimizing temporary and storage allocation (ss 3.2.2, 3.2.3).

   (b) Further evaluation of code generation algorithms (3.4), and their interaction with other phases of an optimizing compiler (3.2).

   (c) Peephole optimization, dealing with optimizations that are best detected after code generation (4.4.2).

   (d) Easing the selection of the "compiler-writer's virtual machine" (3.2.5), to deal with higher-level languages which require considerable run-time support.

(3) New ideas in code generator generation, to deal with:

(a) Optimal case selection in the generation of the code generator tables (4.4.1).

(b) Cases on which the heuristic search could fail due to the complexity of the equivalence, i.e., very high-level machine-operations.

(c) Domains for which axioms were not developed here, e.g., bit field extractions, shifts, and other operations within a data word.

It should be noted that this work is a study to test the feasibility of the approach, not a production implementation. However, a compiler-writing system using this work would be a likely area for future implementation.

In summary, suggested future research includes further work in the direction of the thesis itself, generalizing the machine model and code generator generation schemes, and further work on compilers, particularly on the other phases of compilation as discussed in section 3.2. The present work could also be directly applied to a production implementation, as just mentioned, or to peripheral areas, such as related "automatic generation of..." applications.

Success with the current work suggests that compiler generation, and automatic generation of software from machine descriptions in general, are likely to be developed as practical tools in the not-too-distant future, given sufficient effort on these problems.

Bibliography

Aho, A. V., and Johnson, S.C.: "Optimal Code Generation for Expression Trees", JACM 23, 3 (July 1976), pp 458-501

Allen, F., Carter, J., Harrison, W., Loewner, P., Tapscott, R., Trevillyan, L., Wegman, M.: "The Experimental Compiling Systems Project", IBM Research Report, IBM Yorktown, 1977

Barbacci, M., Barnes, G., Cattell, R., and Siewiorek, D.: ISPS Reference Manual, CMU Computer Science Technical Report, 1978.

Barbacci, M., and Siewiorek, D.: Some Aspects of the Symbolic Manipulation of Computer Descriptions, CMU Computer Science technical report, 1974

Barbacci, M., and Siewiorek, D.: "Evaluation of the CFA Test Programs via Formal Computer Descriptions", Computer 10,10 (October 1977), pp 36-43

Barbacci, M., and Siewiorek, D.: "The CMU RT-CAD System: An Innovative Approach to Computer Aided Design", CMU Computer Science Review 1974-1975

Bell, C. G., and Newell, A.: Computer Structures: Readings and Examples, McGraw-Hill, 1971

Cattell, Roderic G.: "Description of Machine Data Types", internal memo, ISPS group, CMU Computer Science Department, 1976

Cattell, Roderic G.: "A Survey and Critique of Some Models of Code Generation", CMU Computer Science Technical Report, 1977

Coleman, Samuel S.: JANUS: A Universal Intermediate Language, PhD thesis, Electrical Engineering, University of Colorado, 1974

Conway, Melvin E.: "Proposal for an UNCOL", CACM 1,10 (October 1958), pp 5-8

Donegan, Michael K.: An Approach to the Automatic Generation of Code Generators, PhD thesis, Computer Science & Engineering, Rice University, 1973

Ernst, G. W., and Newell, A.: GPS: A Case Study in Generality and Problem Solving, Academic Press, 1969

Elson, M., and Rake, S. T.: "Code-generation Technique for Large-language Compilers", IBM Systems Journal 9,3 (1970), pp 166-188

Feldman, J.: A Formal Semantics for Computer-Oriented Languages, PhD thesis, Computer Science, Carnegie-Mellon University, 1964

Feldman, J. and Gries, D.: "Translator Writing Systems", CACM 11,2 (February 1968) pp 77-113

Fraser, Christopher W.: Automatic Generation of Code Generators, PhD thesis, Computer Science, Yale University, 1977

Glanville, R., and Graham, S.: "A New Method for Compiler Code Generation", Proceedings of the 5th conference on Principles of Programming Languages, 1978 (review of PhD thesis of same title by Glanville, University of California at Berkeley, 1977)

Hobbs, Steven: "Object Code Optimization", thesis proposal, Computer Science, Carnegie-Mellon University, 1976

Knuth, Don: "An Empirical Study of FORTRAN Programs", in Software-Practice and Experience 1, 1971 (pp 105-133)

Leive, Gary: "The Binding of Modules to Abstract Digital Hardware Descriptions", thesis proposal, Electrical Engineering, Carnegie-Mellon University, 1977

McCarthy, J.: "A Basis for a Mathematical Theory of Computation", in Computer Programming and Formal Systems (Eds: Baffort and Hirshberg), North Holland, 1963

McKeeman, W. M., Horning, J. J., and Wortman, D. B.: A Compiler Generator, Prentice Hall, 1970

Miller, Perry L.: Automatic Creation of a Code Generator from a Machine Description, TR-85, Project MAC, Massachusetts Institute of Technology, 1971

Newcomer, Joseph M.: Machine Independent Generation of Optimal Local Code, PhD thesis, Computer Science, Carnegie-Mellon University, 1975

Oakley, John: "Automatic Generation of Diagnostics from ISP", thesis proposal, Computer Science, Carnegie-Mellon University, December 1976

Reiser, J., et al: SAIL, Stanford Artificial Intelligence Lab Memo AIM-289, Computer Science, Stanford University, 1976

Ripken, Knut: Formale Beschreibung von Maschinen, Implementierungen und optimierender

Maschinencodeerzeugung aus attributierten Programmgraphen, dissertation, Technische Universitat Munchen (German) 1977

Samet, Hanan: **Automatically Proving the Correctness of Translations involving Optimized Code,** PhD thesis, Computer Science, Stanford University, 1975

Simoncaux, Donald C.: **High-Level Language Compiling for User-Defineable Architectures,** PhD thesis, Electrical Engineering, Naval Postgraduate School, 1975

Snyder, Alan: **A Portable Compiler for the Language C,** TR-149, Project MAC, Massachusetts Institute of Technology, 1975

Speelpenning, Bert: "A Review of Ripken's Thesis", personal communication from T. Wilcox, 1978

Strong, J., et al: "The Problem of Programming Communication with changing Machines: A Proposed Solution", CACM 1,8 (1958)

Weingart, Steven W.: **An Efficient and Systematic Method of Compiler Code Generation,** PhD thesis, Computer Science, Yale University, 1973

White, John R.: **JOSSLE: A Language for Specifying and Structuring the Semantic Phase of Translators,** PhD thesis, University of California at Santa Barbara, 1973

Wick, John D.: **Automatic Generation of Assemblers,** PhD thesis, Computer Science, Yale University, 1975

Wilcox, Thomas R.: **Generating Machine Code for High-Level Programming Languages,** PhD thesis, Computer Science, Cornell University, 1971

Wulf, W., Johnsson, R., Weinstock, C., Hobbs, S., and Geschke, C.: **The Design of an Optimizing Compiler,** American Elsevier, 1975

Young, Raymond: **The Coder: A Program Module for Code Generation in High-level Language Compilers,** MS thesis, Computer Science, University of Illinois, 1974

Glossary

Access Mode (AM): An expression specifying a location or constant which can be used as an instruction operand. For example, M[C1+R[C2]] where M is memory, R is a register array, and C1 and C2 are constants (this AM represents indexing off a register).

Closed Constant: an integer

Field Assertions: Assertions about instruction field values that result when an instruction field is paired with a field-value list (of the same length). The instruction fields (or OCs) of the instruction format are asserted to have the values specified by the corresponding field-value list elements.

Field-value list: A list of closed constants and parameters.

Instruction Field: A field of the binary instruction representation, e.g., the opcode.

Instruction Format: A list of instruction fields and operand classes. For example, 2-operand instructions on the PDP-11 have the format (OPCODE2 SRC DST) where OPCODE2 is an instruction field, and SRC and DST are operand classes.

Machine-Operation (M-op): An instruction.

Open Constant: represents any closed constant of a given size, i.e., a length in bits is given but no specific value.

Operand Class (OC): A set of access modes, and for each one, field assertions which specify the instruction field values for that access mode (e.g., mode bits, address field).

Operand Computation (OC): Same as Operand Class.

Parameter: variable associated with a leaf of a tree pattern, which is bound to a subtree matched against the pattern tree leaf.

Primary Memory: The memory from which instructions are fetched.

Processor State: The set of all locations which can store values between instruction executions.

Storage Base (SB): A location, or indexed array of locations, in the processor state.

Appendix A: TCOL

The Tree COmmon Language (TCOL) representation of procedural semantics can be thought of as an abstract parse tree. TCOL is used for the representation of the instruction actions in the machine description, the patterns in the code generation tables, and the common language-independent program representation in the compiler.

The TCOL syntax is therefore any convenient representation of a tree. In this thesis, a parenthesized LISP-like notation is used:

```
<tree> ::= <leaf> | (<operator> <tree-list>)
<tree-list> ::= <tree> | <tree-list> <tree>
<leaf> ::= <OC> | <AM> | <constant> | <special leaf>
```

The TCOL semantics are defined in this appendix, by a specification of the TCOL operators.

To define the operators we must first define the four contexts in which an operator tree can occur:

(1) A value context: the tree represents a (bit string) value. It is not necessary to distinguish between different kinds of values (integer, real, etc.) for our purposes.

(2) A boolean context: the tree represents the result TRUE or FALSE.

(3) A statement context: the tree produces no result, it is executed only for the side effects of its subtrees.

(4) A location context: the tree represents a location.

Each operator produces a result in one of these contexts, and similarly demands operands in specific contexts. In the table on the next page, these contexts are indicated in the context column in the form $operand_1$*$operand_2$...=>result where the contexts are represented by V (value), B (boolean), S (statement), and L (location). For example, the context for "←" is given as "L*V=>S", meaning that it is a statement whose operands are a location and a value, respectively. "+", on the other hand, takes two values as arguments, and produces a value as result (V*V=>V). Also note the location descriptor operator, "<>", which is simply used to specify an access to a storage base. Note that a location must be distinguished from a value. The location descriptor operator was found to be more convenient than giving more arguments to the fetch (".") and store ("←") operators (to specify the storage base, etc.)

Pattern trees are also used in this work. For example, machine instructions are represented as patterns and matched against program trees in the code generation process. Pattern trees are defined and represented identically to trees except that pattern leaves may be parameterized for later reference:

AD-A058 872 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
FORMALIZATION AND AUTOMATIC DERIVATION OF CODE GENERATORS.(U)
APR 78 R G CATTELL F44620-73-C-0074
UNCLASSIFIED CMU-CS-78-115 AFOSR-TR-78-1248 NL

2 OF 2
ADA
058 872

END
DATE
FILMED
11-78
DDC

<leaf> ::= $<integer>: <leaf>

The subtree matching this pattern leaf may be referred to elsewhere by "$<integer>". Patterns may also specify leaves that match any node, by omitting the <leaf> altogether and simply writing "$<integer>". See the instruction patterns in appendix B, and the axioms in appendix F, for examples of patterns.

What are patterns to the code generator are trees in the code generator generator. It is therefore necessary to distinguish between "local" parameters in the M-op trees and axioms, and "global" parameters in the global goal tree given to the search routines. Global parameters are distinguished from local ones by doubling the "$":

<leaf> ::= $$<integer>: <leaf> | $$<integer>

It may be necessary to introduce new global parameters when it is necessary to allocate a temporary location or code label in the generation of code for a given tree; these are simply assigned new global parameters in numeric sequence. In the traces in Chapter 4 and appendix D, global parameters are found in the goal trees and code sequences, and local parameters in the axioms and M-ops. The global parameters are then printed out as local parameters when constructing the LOP table for the code generator.

In certain contexts, other special notations are used for TCOL tree leaves. For example, in an access mode tree an open constant (= any constant of size <integer>) is represented by:

<special leaf> ::= #<integer>

Operators	Context	Comments
.	L=>V	dereference operator, "contents of". Will omit this operator unambiguously. by making L=>V conversion implicit.
<>	V☆=>L	location descriptor op; operands are: Storage Base, Index into SB (value), bit position in word, size in bits
+,-,☆,/,MOD	V☆V=>V	integer. NOTE: arithmetic types are machine-defined, these are suggestions.
+S,-S,☆S,/S,	V☆V=>V	short integer arithmetic
+L,-L,☆L,/L,	V☆V=>V	long integer arithmetic
+F,-F,☆F,/F,--F	V☆V=>V	standard floating point arithmetic
+D,-D,☆D,/D,--D	V☆V=>V	double length floating
↑	V☆V=>V	shift left
#FD,#FI,#LD,..	V=>V	Floating, Integer, etc., conversions (eg, #FI=Float->Int)
AND#, OR#, NOT#	V☆V=>V (or V=>V)	Bit-wise logical ops
←#	L☆V=>V	"←" with a value
IF#	B☆V☆V=>V	"IF" with a value
literal	V	occurs as leaf
AND, OR	B☆B=>B	standard conditionals
NOT	B=>B	
EQL, NEQ	V☆V=>B	integer relationals
GTR, LEQ		
LSS, GEQ		
EQLF, ...	V☆V=>B	relationals for floating, etc.
->	B☆L=>S	conditional jump operator
←	L☆V=>S	assignment statement
;	S☆=>S	statement sequence
IF	B☆S☆S=>S	
WHILEDO,DOWHILE	B☆S=>S	
L:	lb☆S=>S	generates label (arg1) before arg2 code
GOTO	V=>S	
CALL,RETURN	V=>S	Procedure linkage
PUSH,	V=>S	
POP	L=>S	

Appendix B: Machine Description Tables

This appendix contains sample MOP machine description tables for those machines used as examples in the thesis: (1) the Mini-S, a PDP-8 slightly simplified for expository purposes, and (2) a PDP-11/20.

The components of the description are described in chapter 2; the syntax for this representation is described here, in a simplified BNF. The syntax for the MOP is relatively straightforward, given the definition of the components and of trees. The only unusual feature is probably that the OCs and M-ops are represented as productions rather than as a simple enumeration of their components. This allows the same syntax to be used for the LOP as for the MOP, because the RHS of the production, which is simply an EMIT node for M-ops (and OCs), can then be any result sequence in the LOP productions.

Notes on BNF: ".." means the preceding non-terminal may be repeated 0 or more times. Items enclosed in curly brackets {..} are comments. Blanks and end of line are not significant. "<>" is the TCOL location-descriptor operator, not a non-terminal of the grammar.

```
<MOP>::= [<I-fld>..] [<SB>..] [<AM>..] [<OC>..] [<Fmt>..] [<M-op>..]
<I-fld>::= ( <I-fld name> <position> <size> <word spec> <field type> )
<field type>::= D | C | D
<SB>::= ( <SB name> <length in words> <length in bits> <SB type> )
<SB type>::= C | G | R | P  {temporary, general, reserved, PC}
<AM>::= <AM name> :: <AM locn tree>
<OC>::= <OC name> : ( <OC-production>.. )
<OC-production>::= <AM name> :: (EMIT[<Fmt#> <timecost> <spacecost>] <field value>..]
<field value>::= <integer> | <parameter label>
<paramter label>::= $<integer>
<Fmt#>::= <integer> {gives # of Fmt in sequence}
<Fmt>::= ( <Fmt item>.. )
<Fmt item>::= <I-fld name> | <OC name>
<M-op>::= <assertion tree> :: (EMIT[<Fmt#> <timecost> <spacecost>] <field value>..]
<assertion tree>::= (; <input/output assertion>..) | <input/output assertion>)
<input/output assertion>::= (IF <input assertion> <output assertion>) | <output assertion>
<input assertion>::= <bool tree>
<output assertion>::= No.Op | (← <locn spec> <expr tree>)
<locn spec>::= <AM name> | <OC name>
<expr tree>::= (<value-operator> <expr tree>..) | <locn tree>
<bool tree>::= (<boolean-operator> <expr tree>..) | <leaf>
<locn tree>::= (<> <SB name> <expr tree> <position> <size>) | <leaf>

<leaf> may be different things depending on the context.  In an M-op:
<leaf>::= <constant> | <OC name> | <AM name>
In an AM (access mode):
<leaf>::= <constant> | <open constant>
<open constant>::= <parameter label> : #<integer> {reprsnts const with <integer> bits}
In a goal tree:
<leaf>::= <constant> | <OC name> | <AM name> | <wild node>
<wild node>::= <parameter label>
```

```
!MOP file for Mini-S (simplified PDP-8)!


!I-flds! [
(OP 0 3 0 O)
(I.BIT 3 1 0 C)
(ADR 4 8 0 D)
(IO.BITS 4 8 0 D)
(UBITS 5 7 0 O)
(UCLASS 4 1 0 O) ]

!SBs! [
(PC 1 8 P)
(Mp 256 12 M)
(Acc 1 12 G)
(IO.REG 1 8 R)
(L 1 8 R) ]

!AMs! [
%8:      $1:#8
%Mp:     (<> Mp $1:#8 0 12)
%eMp:    (<> Mp (<> Mp $1:#8 0 12) 0 12)
%PC:     (<> PC 1 0 8)
%Acc:    (<> Acc 1 0 12)
%L:      (<> L 1 0 8)
%IO.REG: (<> IO.REG 1 0 8) ]

!OCs! [

Y: (
%8 :: (EMIT[5 0 0]  $1 0)
%Mp ::  (EMIT[5 1 0]  $1 1) )

Z: (
%Mp :: (EMIT[5 1 0]  $1 0)
%eMp ::  (EMIT[5 2 0]  $1 1) )

IO: (
%8 :: (EMIT[6 0 0]  $1) ) ]

[
!I-FMTs!
!FMT 1! (OP Z)             !1-opnd format!
!FMT 2! (OP Y)             !jump format!
!FMT 3! (OP UCLASS UBITS)  !micro format!
!FMT 4! (OP IO)            !IOT format!

!OC-FMTs!
!FMT 5! (ADR I.BIT)        !Y and Z!
!FMT 6! (IO.BITS) ]        !IO!


!Mops! [

(← %Acc (AND %Acc $1:Z)) ::
(EMIT[AND 1 1 1]  0 $1)

(← %Acc (+ %Acc $1:Z)) ::
(EMIT[TAD 1 1 1]  1 $1)
```

```
(; (← $1:Z (+ $1:Z 1)) (=> (EQL $1:Z -1) (← %PC (+ %PC 1)))) ::
(EMIT[ISZ 1 1 1]  2 $1)

(; (← $1:Z %Acc) (← %Acc 0)) ::
(EMIT[DCA 1 1 1]  3 $1)

(; (← %L %PC) (← %PC $1:Y)) ::
(EMIT[JMS 2 1 1]  4 $1)

(← %PC $1:Y) ::
(EMIT[JMP 2 1 1]  5 $1)

(← %IO.REG IO) ::
(EMIT[IOT 4 1 1]  6 $1)

(← %Acc (NOT %Acc)) ::
(EMIT[COMA 3 1 1]  7 0 40)

(← %Acc 0) ::
(EMIT[CLRA 3 1 1]  7 0 20)

(← %Acc (+ %Acc 1)) ::
(EMIT[INCA 3 1 1]  7 0 10)

(← %Acc (- %Acc 1)) ::
(EMIT[DECA 3 1 1]  7 0 4)

(← %Acc (↑ %Acc 1)) ::
(EMIT[SLA 3 1 1]  7 0 1)

(NO.OP) ::
(EMIT[NOP 3 1 1]  7 0 0)

(← %Acc 1) ::
(EMIT[SET1A 3 1 1]  7 0 30)

(← %Acc 2) ::
(EMIT[SET2A 3 1 1]  7 0 31)

(← %PC %L) ::
(EMIT[RTS 3 1 1]  7 1 40)

(← %PC %Acc) ::
(EMIT[JMPA 3 1 1]  7 1 20)

(=> (LSS %Acc 0) (← %PC (+ %PC 1))) ::
(EMIT[SKPL 3 1 1]  7 1 4)

(=> (EQL %Acc 0) (← %PC (+ %PC 1))) ::
(EMIT[SKPE 3 1 1]  7 1 5)

(=> (NEQ %Acc 0) (← %PC (+ %PC 1))) ::
(EMIT[SKPNE 3 1 1]  7 1 2)

(=> (GTR %Acc 0) (← %PC (+ %PC 1))) ::
(EMIT[SKPG 3 1 1]  7 1 1)

(=> (LEQ %Acc 0) (← %PC (+ %PC 1))) ::
```

```
(EMIT[SKPLE 3 1 1]  7 1 6)

(=> (GEQ %Acc 0) (<- %PC (+ %PC 1))) ;;
(EMIT[SKPGE 3 1 1]  7 1 3) ]
```

```
!MOP file for PDP11/20 (truncated)!


!I-flds! [
(OpCode1 0 10 0 O)  !1-opnd!
(OpCode2 0 4 0 O)   !2-opnd!
(OpCode8 0 8 0 O)   !branch!
(OpCodeJ 0 7 0 O)   !jump!
(OpCodeR 0 13 0 O)  !return!

(OffsetB 8 8 0 D)   !branch!
(SrcMode 4 3 0 D)   !2-opnd!
(SrcReg 7 3 0 D)    !2-opnd!
(SrcIndex 0 16 1 D) !2-opnd!
(DstMode 10 3 0 C)  !1-opnd & 2-opnd!
(DstReg 13 3 0 D)   !1-opnd & 2-opnd!
(DstIndex 0 16 2 D) !1-opnd & 2-opnd! ]


!SBs! [
(M 65536 8 M)
(N 1 1 C)
(Z 1 1 C)
(V 1 1 C) !not used!
(C 1 1 C) !not used!
(PC 1 16 P)
(SP 1 16 R)
(R  6 16 G) ]


!AMs! [
%8: $1:#8
%16: $1:#16
%SP: (<> SP 0 0 16)
%PC: (<> PC 0 0 16)
%N: (<> N 0 0 1)
%Z: (<> Z 0 0 1)
%C: (<> C 0 0 1)
%L: (<> M (+ %PC (↑ $1:#8 1)) 0 16)
%R:  (<> R $1:#3 0 16)
%Rb: (<> R $1:#3 8 8)
%M:  (<> M $1:#16 0 16)
%Mb: (<> M $1:#16 0 8)
%eR:  (<> M (<> R $1:#3 0 16) 0 16)
%eRb: (<> M (<> R $1:#3 0 16) 8 8)
%eM:  (<> M (<> M $1:#16 0 16) 0 16)
%eMb: (<> M (<> M $1:#16 0 16) 0 8)
%R+C:   (<> M (+ (<> R $1:#3 0 16) $2:#16) 0 16)
%R+Cb:  (<> M (+ (<> R $1:#3 0 16) $2:#16) 0 8)
%eR+C:  (<> M (<> M (+ (<> R $1:#3 0 16) $2:#16) 0 16) 0 16)
%eR+Cb: (<> M (<> M (+ (<> R $1:#3 0 16) $2:#16) 0 16) 0 8)
!auto-increment and -decrement not used except with PC!
%+R: (<> M (← %R (+ %R 2)) 0 16)
%+Rb: (<> M (← %R (+ %R 1)) 0 8)
%-R: (<> M (← %R (- %R 2)) 0 16)
%-Rb: (<> M (← %R (- %R 1)) 0 8) ]


!OCs! [
```

```
Src: (
%16 ::  (EMIT[13 0.0 1] 2 7 $1)
%R ::   (EMIT[12 0.0 0] 0 $1)
%M ::   (EMIT[13 1.5 1] 3 7 $1)
%eR ::  (EMIT[12 1.5 0] 1 $1)
%eM ::  (EMIT[13 2.7 1] 7 7 $1)
%R+C :: (EMIT[13 2.7 1] 6 $1 $2)
%eR+C:: (EMIT[13 3.9 1] 7 $1 $2) )

SrcB: (
%Rb::   (EMIT[12 0.0 0] 0 $1)
%Mb::   (EMIT[13 1.5 1] 3 7 $1)
%eRb::  (EMIT[12 1.5 0] 1 $1)
%eMb::  (EMIT[13 2.7 1] 7 7 $1)
%R+Cb:: (EMIT[13 2.7 1] 6 $1 $2)
%eR+Cb::(EMIT[13 3.9 1] 7 $1 $2) )

Dst: (
%16 ::  (EMIT[10 0.0 1] 2 7 $1)
%R ::   (EMIT[9  0.0 0] 0 $1)
%M ::   (EMIT[10 1.4 1] 3 7 $1)
%eR ::  (EMIT[9  1.4 0] 1 $1)
%eM ::  (EMIT[10 2.6 1] 7 7 $1)
%R+C :: (EMIT[10 2.6 1] 6 $1 $2)
%eR+C:: (EMIT[10 3.8 1] 7 $1 $2) )

DstB: (
%Rb::   (EMIT[9  0.0 0] 0 $1)
%Mb::   (EMIT[10 1.4 1] 3 7 $1)
%eRb::  (EMIT[9  1.4 0] 1 $1)
%eMb::  (EMIT[10 2.6 1] 7 7 $1)
%R+Cb:: (EMIT[10 2.6 1] 6 $1 $2)
%eR+Cb::(EMIT[10 3.8 1] 7 $1 $2) )

SrcR: ( !for JUMP, JSR!
%R :: (EMIT[11 0.0 0] $1) )

DstR: ( !for RTS!
%R :: (EMIT[8 0.0 0] $1) )

Adr: ( !for branches!
%L :: (EMIT[14 0.0 0] $1) ) ]


!I-fmts! [
!1! (OpCode1 Dst)
!2! (OpCode2 Src Dst)
!3! (OpCodeB Adr)
!4! (OpCodeJ SrcR Dst)
!5! (OpCodeR DstR)
!6! (Opcode1 DstB)
!7! (OpCode2 SrcB DstB)
!OC-fmts!
!8! (DstReg)
!9! (DstMode DstReg)
!10! (DstMode DstReg DstIndex)
!11! (SrcReg)
!12! (SrcMode SrcReg)
!13! (SrcMode SrcReg SrcIndex)
```

```
!14! (Offset8) ]


!M-ops! [

!byte versions of instructions have been omitted; these are
 similar to word instrs but use OCs Src8 and Dst8 and formats 6 and 7!

!1-opnd instructions!
(; (← $1:DST 0) (← ZN (LSS 0 0)) (← ZZ (EQL 0 0))) ::
(EMIT(CLR 1 1 1) 50 $1)

(; (← $1:DST (NOT $1:DST)) (← ZN (LSS (NOT $1:DST) 0)) (← ZZ (EQL (NOT $1:DST) 0))) ::
(EMIT(COM 1 1 1) 51 $1)

(; (← $1:DST (+ $1:DST 1)) (← ZN (LSS (+ $1:DST 1) 0)) (← ZZ (EQL (+ $1:DST 1) 0))) ::
(EMIT(INC 1 1 1) 52 $1)

(; (← $1:DST (- $1:DST 1)) (← ZN (LSS (- $1:DST 1) 0)) (← ZZ (EQL (- $1:DST 1) 0))) ::
(EMIT(DEC 1 1 1) 53 $1)

(; (← $1:DST (- $1:DST)) (← ZN (GEQ $1:DST 0)) (← ZZ (EQL $1:DST 0))) ::
(EMIT(NEG 1 1 1) 54 $1)

(; (← ZN (LSS $1:DST 0)) (← ZZ (EQL $1:DST 0))) ::
(EMIT(TST 1 1 1) 57 $1)

(; (← $1:DST (↑ $1:DST -1)) (← ZN (LSS (↑ $1:DST -1) 0)) (← ZZ (EQL (↑ $1:DST -1) 0))) ::
(EMIT(ASR 1 1 1) 62 $1)

(; (← $1:DST (↑ $1:DST 1)) (← ZN (LSS (↑ $1:DST 1) 0)) (← ZZ (EQL (↑ $1:DST 1) 0))) ::
(EMIT(ASL 1 1 1) 63 $1)


!2-opnd instructions!
(; (← $1:DST $2:SRC) (← ZN (LSS $2:SRC 0)) (← ZZ (EQL $2:SRC 0))) ::
(EMIT(MOV 2 1 1) 1 $2 $1)

(; (← $1:DST (+ $1:DST $2:SRC)) (← ZN (LSS (+ $1:DST $2:SRC) 0)) (← ZZ (EQL (+ $1:DST $2:SRC) 0)))
(EMIT(ADD 2 1 1) 6 $2 $1)

(; (← $1:DST (- $1:DST $2:SRC)) (← ZN (LSS $1:DST $2:SRC)) (← ZZ (EQL $1:DST $2:SRC))) ::
(EMIT(SUB 2 1 1) 16 $2 $1)

(; (← ZN (LSS $2:DST $1:SRC)) (← ZZ (EQL $2:DST $1:SRC))) ::
(EMIT(CMP 2 1 1) 2 $2 $1)

(; (← ZN (LSS (OR $1:DST $2:SRC) 0)) (← ZZ (EQL (OR $1:DST $2:SRC) 0))) ::
(EMIT(BIT 2 1 1) 3 $2 $1)

(; (← $1:DST (AND $1:DST (NOT $2:SRC)))
   (← ZN (LSS (AND $1:DST (NOT $2:SRC)) 0)) (← ZZ (EQL (AND $1:DST (NOT $2:SRC)) 0))) ::
(EMIT(BIC 2 1 1) 4 $2 $1)

(; (← $1:DST (OR $1:DST $2:SRC))
   (← ZN (LSS (OR $1:DST $2:SRC) 0)) (← ZZ (EQL (OR $1:DST $2:SRC) 0))) ::
(EMIT(BIS 2 1 1) 5 $2 $1)

!subroutine and jump instructions!
```

```
(CALL) ::
(EMIT[JSR 4 1 1] 4 7 8)

(RETURN) ::
(EMIT[RTS 5 1 1] 20 7)

(← %PC $1:#16) ::
(EMIT[JMP 4 1 1] 0 1 $1)

|branch instructions|

(← %PC $1:ADR) ::
(EMIT[BR 3 1 1] 4 $1)

(-> %Z $1:ADR) ::
(EMIT[BEQ 3 1 1] 14 $1)

(-> (NOT %Z) $1:ADR) ::
(EMIT[BNE 3 1 1] 10 $1)

(-> %N $1:ADR) ::
(EMIT[BLT 3 1 1] 24 $1)

(-> (NOT %N) $1:ADR) ::
(EMIT[BGE 3 1 1] 20 $1)

(-> (OR %Z %N) $1:ADR) ::
(EMIT[BLE 3 1 1] 34 $1)

(-> (AND (NOT %Z) (NOT %N)) $1:ADR) ::
(EMIT[BGT 3 1 1] 30 $1)

]
```

Appendix C: Code Generator Prototype Trace

This is a trace of a user's interaction with the stand-alone code generator, CODE, whose tables have been filled in for the PDP-10. (Other phases of the PQCC compiler have not been completed at the time of this writing.) The user's routine, called TEST, is printed out in the parenthesized TCOL notation. The TEST program is to add the integers up to 10 and print the result. TEST.TCL, the TCOL file, is read by CODE to produce TEST.MAC, in assembly language. Then, TEST.MAC is assembled and loaded with the user's main program, written in BLISS-10, and executed. TEST prints out the answer that the sum is 45.

```
.SET WATCH DAY,RUN
.type test.tcl
[13:05:18]
(BLOCK

X(OWN)
R1(REGISTER 1)
PRNT(EXTERNAL)
TEST(GLOBAL ROUTINE)

(ROUTINE TEST (;
        (← X 0)
        (← R1 1)
        (DOUNTIL
                (;
                        (← X (+ (. X) (. R1)))
                        (← R1 (+ (. R1) 1)) )
                (EQL (. R1) 10) )
        (PUSH (. X))
        (CALL PRNT)
        (POP X) )) )
[0.23   ]

.ru code
[13:05:19]

Input File: test.tcl

Output File: test.mac

CODE pass 0:
CODE pass 1:
CODE pass 2:

EXIT
[0.26   ]

.type test.mac
[13:05:21]
X:      OCT     0
        EXTERN  PRNT
        INTERN  TEST
TEST:   SETZM   X
        MOVEI   1,1
L0:     ADDM    1,X
        MOVEI   1,1(1)
```

```
        CAIE    1,12
        JRST    L0
        JRST    L1
L1:     PUSH    X
        PUSHJ   PRNT
        POP     X
        POPJ    0
        END
[0.08   ]

.type main.bli
[13:05:21]
module main(stack)=
begin
external test,outnum,outmsg;

global routine prnt(y)=outnum(-1,.y,10,0);

outmsg(-1,plit asciz 'CODE test program?M?JX=');
test()
end;

[0.06   ]

.ex main.bli,test.mac,sys:blilib/lib
[13:05:22]
BLIS10: MAIN.BLI
MACRO: .MAIN
LINK:  Loading
[LNKXCT MAIN Execution]
CODE test program
X=45
EXIT
[1.03   ]
```

Appendix D: Search Examples

```
Search: (← ZACC ZMP)
Attempting M-op-match
Attempting Decompositions
Attempting Transformations
 Feasible[1]: (← ZACC (+ ZACC $1:Z))
  Transform: (← ZACC ZMP) => (← ZACC (+ ZACC $1:Z))
   Transform: ZACC => ZACC
   Transform: ZMP => (+ ZACC $1:Z)
    Applying $1 :: (+ 0 $1)  to: ZMP
     Transform: (+ 0 ZMP) => (+ ZACC $1:Z)
      Transform: 0 => ZACC
       Applying Fetch Decomposition to: 0 using: $$1:ZACC
        Search: (← $$1:ZACC 0)
        Attempting M-op-match
        M-op Match: (; (ALLOC $$2:ZMP) (EMIT[DCA 1 1 1] 3 $$2:ZMP))
        M-op Match: (EMIT[CLRA 3 1 1] 7 0 20)
      Transform: ZMP => $1:Z
 Feasible[2]: (← ZACC (+ ZACC 1))
  Transform: (← ZACC ZMP) => (← ZACC (+ ZACC 1))
   Transform: ZACC => ZACC
   Transform: ZMP => (+ ZACC 1)
    Applying $1 :: (+ 0 $1)  to: ZMP
     Transform: (+ 0 ZMP) => (+ ZACC 1)
      Transform: 0 => ZACC
       Applying Fetch Decomposition to: 0 using: $$3:ZACC
        Search: (← $$3:ZACC 0)
        Attempting M-op-match
        M-op Match: (; (ALLOC $$4:ZMP) (EMIT[DCA 1 1 1] 3 $$4:ZMP))
        M-op Match: (EMIT[CLRA 3 1 1] 7 0 20)
      Transform: ZMP => 1
       [fail on ZMP ]
      [fail on ZMP ]
      Applying (+ $1 $2) :: (+ $2 $1)  to: (+ 0 ZMP)
       Transform: (+ ZMP 0) => (+ ZACC 1)
        Transform: ZMP => ZACC
         Depth Limit Reached
         [fail on ZMP ]
        [fail on ZMP ]
        Depth Limit Reached
        [fail on (+ ZMP 0) ]
       [fail on (+ ZMP 0) ]
      [fail on (+ 0 ZMP) ]
     [fail on (+ 0 ZMP) ]
    [fail on ZMP ]
   [fail on ZMP ]
   [fail on (← ZACC ZMP) ]
 [fail on (← ZACC ZMP) ]
[fail on (← ZACC ZMP) ]
Feasible[3]: (← ZACC (- ZACC 1))
 Transform: (← ZACC ZMP) => (← ZACC (- ZACC 1))
  Transform: ZACC => ZACC
  Transform: ZMP => (- ZACC 1)
   Applying $1 :: (- (- $1))  to: ZMP
    Transform: (- (- ZMP)) => (- ZACC 1)
     Applying (- $1) :: (- 0 $1)  to: (- (- ZMP))
```

```
      Transform: (- 0 (- ZMP)) => (- ZACC 1)
       Transform: 0 => ZACC
        Depth Limit Reached
        [fail on 0 ]
       [fail on 0 ]
       Depth Limit Reached
       [fail on (- 0 (- ZMP)) ]
      [fail on (- 0 (- ZMP)) ]
     [fail on (- (- ZMP)) ]
    [fail on (- (- ZMP)) ]
   [fail on ZMP ]
  [fail on ZMP ]
  [fail on (← ZACC ZMP) ]
 [fail on (← ZACC ZMP) ]
[fail on (← ZACC ZMP) ]
Breadth Limit Reached(24)
------------------------
Nodes Examined: 25
Est. Seconds: .756@-1
Result Sequence(s):
      [Alloc $$1:ZACC]
<<<<<<<<<<<<<<<<<<<<<<<<<
      [Alloc $$2:ZMP]
      DCA     $$2:ZMP
-----------or-----------
      CLRA
>>>>>>>>>>>>>>>>>>>>>>>>>
      TAD     ZMP
Best Sequence is:
      [Alloc $$1:ZACC]
      CLRA
      TAD     ZMP
------------------------

Search: (← ZACC (- ZACC))
Attempting M-op-match
Attempting Decompositions
Attempting Transformations
 Feasible[1]: (← ZACC (- ZACC 1))
  Transform: (← ZACC (- ZACC)) => (← ZACC (- ZACC 1))
   Transform: ZACC => ZACC
   Transform: (- ZACC) => (- ZACC 1)
    Applying (- $1) :: (- 0 $1)  to: (- ZACC)
     Transform: (- 0 ZACC) => (- ZACC 1)
      Transform: 0 => ZACC
       Applying Fetch Decomposition to: 0 using: $$1:ZACC
        Search: (← $$1:ZACC 0)
        Attempting M-op-match
        M-op Match: (; (ALLOC $$2:ZMP) (EMIT[DCA 1 1 1] 3 $$2:ZMP))
        M-op Match: (EMIT[CLRA 3 1 1] 7 0 20)
      Transform: ZACC => 1
       [fail on ZACC ]
      [fail on ZACC ]
      [fail on (- 0 ZACC) ]
     [fail on (- 0 ZACC) ]
    [fail on (- ZACC) ]
   [fail on (- ZACC) ]
   [fail on (← ZACC (- ZACC)) ]
```

```
 [fail on (← ZACC (- ZACC)) ]
[fail on (← ZACC (- ZACC)) ]
Feasible[2]: (← ZACC (+ ZACC $1:Z))
 Transform: (← ZACC (- ZACC)) => (← ZACC (+ ZACC $1:Z))
  Transform: ZACC => ZACC
  Transform: (- ZACC) => (+ ZACC $1:Z)
   Applying (- $1) :: (+ 0 (- $1))  to: (- ZACC)
    Transform: (+ 0 (- ZACC)) => (+ ZACC $1:Z)
     Transform: 0 => ZACC
      Applying Fetch Decomposition to: 0 using: $$3:ZACC
       Search: (← $$3:ZACC 0)
       Attempting M-op-match
       M-op Match: (; (ALLOC $$4:ZMP) (EMIT[DCA 1 1 1] 3 $$4:ZMP))
       M-op Match: (EMIT[CLRA 3 1 1] 7 0 20)
     Transform: (- ZACC) => $1:Z
      Applying Fetch Decomposition to: (- ZACC) using: $$5:ZMP
       Search: (← $$5:ZMP (- ZACC))
       Attempting M-op-match
       Attempting Decompositions
       Attempting Transformations
        Feasible[1]: (← $1:Z (+ $1:Z 1))
         Transform: (← $$5:ZMP (- ZACC)) => (← $1:Z (+ $1:Z 1))
          Transform: $$5:ZMP => $1:Z
          Transform: (- ZACC) => (+ $1:Z 1)
           Depth Limit Reached
           [fail on (- ZACC) ]
          [fail on (- ZACC) ]
          Depth Limit Reached
          [fail on (← $$5:ZMP (- ZACC)) ]
         [fail on (← $$5:ZMP (- ZACC)) ]
        [fail on (← $$5:ZMP (- ZACC)) ]
        Feasible[2]: (← $1:Z ZACC)
         Transform: (← $$5:ZMP (- ZACC)) => (← $1:Z ZACC)
          Transform: $$5:ZMP => $1:Z
          Transform: (- ZACC) => ZACC
           Depth Limit Reached
           [fail on (- ZACC) ]
          [fail on (- ZACC) ]
          Depth Limit Reached
          [fail on (← $$5:ZMP (- ZACC)) ]
         [fail on (← $$5:ZMP (- ZACC)) ]
        [fail on (← $$5:ZMP (- ZACC)) ]
        Feasible[3]: (← ZACC (- ZACC 1))
        Attempting Store-Decomposition using: $$6:ZACC
         Search: (← $$5:ZMP $$6:ZACC)
         Attempting M-op-match
         M-op Match: (; (ALLOC $$7:ZACC) (EMIT[DCA 1 1 1] 3 $$5:ZMP))
         Transform: (- ZACC) => (- ZACC 1)
          Depth Limit Reached
          [fail on (- ZACC) ]
         [fail on (- ZACC) ]
        [fail on (← $$5:ZMP (- ZACC)) ]
       Breadth Limit Reached(9)
       [fail on (← $$5:ZMP (- ZACC)) ]
      [fail on (- ZACC) ]
     [fail on (- ZACC) ]
     Applying (+ $1 $2) :: (+ $2 $1)  to: (+ 0 (- ZACC))
      Transform: (+ (- ZACC) 0) => (+ ZACC $1:Z)
       Transform: (- ZACC) => ZACC
```

```
        Applying Fetch Decomposition to: (- ZACC) using: $$8:ZACC
         Search: (← $$8:ZACC (- ZACC))
         Attempting M-op-match
         Depth Limit Reached
         [fail on (← $$8:ZACC (- ZACC)) ]
        [fail on (- ZACC) ]
       [fail on (- ZACC) ]
       Applying (+ $1 $2) :: (+ $2 $1)  to: (+ (- ZACC) 0)
        Transform: (+ 0 (- ZACC)) => (+ ZACC $1:Z)
         Transform: 0 => ZACC
          Depth Limit Reached
          [fail on 0 ]
         [fail on 0 ]
         Depth Limit Reached
         [fail on (+ 0 (- ZACC)) ]
        [fail on (+ 0 (- ZACC)) ]
       [fail on (+ (- ZACC) 0) ]
      [fail on (+ (- ZACC) 0) ]
     [fail on (+ 0 (- ZACC)) ]
    [fail on (+ 0 (- ZACC)) ]
   Applying (- $1) :: (+ (NOT $1) 1)  to: (- ZACC)
    Transform: (+ (NOT ZACC) 1) => (+ ZACC $1:Z)
     Transform: (NOT ZACC) => ZACC
      Applying Fetch Decomposition to: (NOT ZACC) using: $$9:ZACC
       Search: (← $$9:ZACC (NOT ZACC))
       Attempting M-op-match
       M-op Match: (EMIT[COMA 3 1 1] 7 0 40)
     Transform: 1 => $1:Z
      Applying Constant Fetch Decomposition
Feasible[3]: (← ZACC (+ ZACC 1))
 Transform: (← ZACC (- ZACC)) => (← ZACC (+ ZACC 1))
  Transform: ZACC => ZACC
  Transform: (- ZACC) => (+ ZACC 1)
   Applying (- $1) :: (+ 0 (- $1))  to: (- ZACC)
    Transform: (+ 0 (- ZACC)) => (+ ZACC 1)
     Transform: 0 => ZACC
      Applying Fetch Decomposition to: 0 using: $$11:ZACC
       Search: (← $$11:ZACC 0)
       Attempting M-op-match
       M-op Match: (; (ALLOC $$12:ZMP) (EMIT[DCA 1 1 1] 3 $$12:ZMP))
       M-op Match: (EMIT[CLRA 3 1 1] 7 0 20)
     Transform: (- ZACC) => 1
      [fail on (- ZACC) ]
     [fail on (- ZACC) ]
     Applying (+ $1 $2) :: (+ $2 $1)  to: (+ 0 (- ZACC))
      Transform: (+ (- ZACC) 0) => (+ ZACC 1)
       Transform: (- ZACC) => ZACC
        Applying Fetch Decomposition to: (- ZACC) using: $$13:ZACC
         Search: &
         Attempting M-op-match
         Depth Limit Reached
         [fail on & ]
        [fail on (- ZACC) ]
       [fail on (- ZACC) ]
       Applying (+ $1 $2) :: (+ $2 $1)  to: (+ (- ZACC) 0)
        Transform: (+ 0 (- ZACC)) => (+ ZACC 1)
         Transform: 0 => ZACC
          Depth Limit Reached
          [fail on 0 ]
```

```
          [fail on 0 ]
          Depth Limit Reached
          [fail on (+ 0 (- ZACC)) ]
         [fail on (+ 0 (- ZACC)) ]
        [fail on (+ (- ZACC) 0) ]
       [fail on (+ (- ZACC) 0) ]
      [fail on (+ 0 (- ZACC)) ]
     [fail on (+ 0 (- ZACC)) ]
    Applying (- $1) :: (+ (NOT $1) 1)  to: (- ZACC)
     Transform: (+ (NOT ZACC) 1) => (+ ZACC 1)
      Transform: (NOT ZACC) => ZACC
       Applying Fetch Decomposition to: (NOT ZACC) using: $$14:ZACC
        Search: (← $$14:ZACC (NOT ZACC))
        Attempting M-op-match
        M-op Match: (EMIT[COMA 3 1 1] 7 0 40)
       Transform: 1 => 1
  Feasible[4]: (← $1:Z (+ $1:Z 1))
  Attempting Store-Decomposition using: $$15:ZMP
   Search: (← ZACC $$15:ZMP)
   Attempting M-op-match
   Depth Limit Reached
   [fail on (← ZACC $$15:ZMP) ]
  [fail on (← ZACC (- ZACC)) ]
  Feasible[5]: (← $1:Z ZACC)
  Attempting Store-Decomposition using: $$16:ZMP
   Search: (← ZACC $$16:ZMP)
   Attempting M-op-match
   Depth Limit Reached
   [fail on (← ZACC $$16:ZMP) ]
  [fail on (← ZACC (- ZACC)) ]
 No more feasible M-ops
------------------------
Nodes Examined: 58
Est. Seconds:  .199
Result Sequence(s):
<<<<<<<<<<<<<<<<<<<<<<<<<
       [Alloc $$9:ZACC]
       COMA
       [Const $$10:ZMP 1]
       TAD     $$10:ZMP
-----------or-----------
       [Alloc $$14:ZACC]
       COMA
       INCA
>>>>>>>>>>>>>>>>>>>>>>>>>
Best Sequence is:
       [Alloc $$14:ZACC]
       COMA
       INCA
------------------------

 Search: (← ZACC (+ ZACC $$1:Z))
 Attempting M-op-match
 M-op Match: (EMIT[TAD 1 1 1] 1 $$1:Z)
------------------------
Nodes Examined: 1
Est. Seconds:  .372e-2
Result Sequence(s):
```

```
      TAD     $$1:Z
Best Sequence is:
      TAD     $$1:Z
------------------------

Search: (← ZACC (- ZMP ZACC))
Attempting M-op-match
Attempting Decompositions
Attempting Transformations
 Feasible[1]: (← ZACC (- ZACC 1))
  Transform: (← ZACC (- ZMP ZACC)) => (← ZACC (- ZACC 1))
   Transform: ZACC => ZACC
   Transform: (- ZMP ZACC) => (- ZACC 1)
    Transform: ZMP => ZACC
     Applying Fetch Decomposition to: ZMP using: $$1:ZACC
      Search: (← $$1:ZACC ZMP)
      Attempting M-op-match
      M-op Match: (; (ALLOC $$2:ZACC) (EMIT[CLRA 3 1 1] 7 0 20) (EMIT[TAD 1 1 1] 1 ZMP))
    Transform: ZACC => 1
     [fail on ZACC ]
    [fail on ZACC ]
    [fail on (- ZMP ZACC) ]
   [fail on (- ZMP ZACC) ]
   [fail on (← ZACC (- ZMP ZACC)) ]
  [fail on (← ZACC (- ZMP ZACC)) ]
 [fail on (← ZACC (- ZMP ZACC)) ]
 Feasible[2]: (← ZACC (- ZACC))
  Transform: (← ZACC (- ZMP ZACC)) => (← ZACC (- ZACC))
   Transform: ZACC => ZACC
   Transform: (- ZMP ZACC) => (- ZACC)
    [fail on (- ZMP ZACC) ]
   [fail on (- ZMP ZACC) ]
   [fail on (← ZACC (- ZMP ZACC)) ]
  [fail on (← ZACC (- ZMP ZACC)) ]
 [fail on (← ZACC (- ZMP ZACC)) ]
 Feasible[3]: (← ZACC (+ ZACC $1:Z))
  Transform: (← ZACC (- ZMP ZACC)) => (← ZACC (+ ZACC $1:Z))
   Transform: ZACC => ZACC
   Transform: (- ZMP ZACC) => (+ ZACC $1:Z)
    Applying (- $1 $2) :: (+ $1 (- $2))  to: (- ZMP ZACC)
     Transform: (+ ZMP (- ZACC)) => (+ ZACC $1:Z)
      Transform: ZMP => ZACC
       Applying Fetch Decomposition to: ZMP using: $$4:ZACC
       Search: (← $$4:ZACC ZMP)
        Attempting M-op-match
        M-op Match: (; (ALLOC $$5:ZACC) (EMIT[CLRA 3 1 1] 7 0 20) (EMIT[TAD 1 1 1] 1 ZMP))
      Transform: (- ZACC) => $1:Z
       Applying Fetch Decomposition to: (- ZACC) using: $$6:ZMP
        Search: (← $$6:ZMP (- ZACC))
        Attempting M-op-match
        Attempting Decompositions
        Attempting Transformations
         Feasible[1]: (← $1:Z (+ $1:Z 1))
          Transform: (← $$6:ZMP (- ZACC)) => (← $1:Z (+ $1:Z 1))
           Transform: $$6:ZMP => $1:Z
           Transform: (- ZACC) => (+ $1:Z 1)
            Depth Limit Reached
            [fail on (- ZACC) ]
```

```
          [fail on (- ZACC) ]
          Depth Limit Reached
          [fail on (← $$6:ZMP (- ZACC)) ]
         [fail on (← $$6:ZMP (- ZACC)) ]
        [fail on (← $$6:ZMP (- ZACC)) ]
       Breadth Limit Reached(3)
       [fail on (← $$6:ZMP (- ZACC)) ]
      [fail on (- ZACC).]
     [fail on (- ZACC) ]
     Applying (+ $1 $2) :: (+ $2 $1)  to: (+ ZMP (- ZACC))
      Transform: (+ (- ZACC) ZMP) => (+ ZACC $1:Z)
       Transform: (- ZACC) => ZACC
        Applying Fetch Decomposition to: (- ZACC) using: $$7:ZACC
         Search: (← $$7:ZACC (- ZACC))
         Attempting M-op-match
         M-op Match: (; (ALLOC $$8:ZACC) (EMIT[COMA 3 1 1] 7 0 40) (EMIT[INCA 3 1 1] 7 0 10))
       Transform: ZMP => $1:Z
Breadth Limit Reached(23)
-----------------------
Nodes Examined: 24
Est. Seconds:  899@-1
Result Sequence(s):
      [Alloc $$7:ZACC]
      [Alloc $$8:ZACC]
      COMA
      INCA
      TAD     ZMP
Best Sequence is:
      [Alloc $$7:ZACC]
      [Alloc $$8:ZACC]
      COMA
      INCA
      TAD     ZMP
-----------------------

Search: (IF (EQL ZACC 0) (← ZACC 1))
Attempting M-op-match
Attempting Decompositions
Applying  (IF $1 $2) :: (; (-> (NOT $1) $3:ZMP) $2 (LABEL $3:ZMP))
 Simplifying (NOT (EQL ZACC 0)) to (NEQ ZACC 0)
 Search: (; (-> (NEQ ZACC 0) $$1:ZMP) (← ZACC 1) (LABEL $$1:ZMP))
 Attempting M-op-match
 Attempting Decompositions
 Applying Sequence-Decomposition
  Search: (-> (NEQ ZACC 0) $$1:ZMP)
  Attempting M-op-match
  Attempting Decompositions
  Applying Skip-Decomposition
   Search: (GOTO $$1:ZMP)
   Attempting M-op-match
   Attempting Decompositions
   Applying  (GOTO $1) :: (← ZPC $1)
    Search: (← ZPC $$1:ZMP)
    Attempting M-op-match
    M-op Match: (EMIT[JMP 1 1 1] 5 $$1:ZMP)
   Attempting Transformations
   Simplifying (NOT (NEQ ZACC 0)) to (EQL ZACC 0)
   Search: (-> (EQL ZACC 0) (+ ZPC 1))
```

```
    Attempting M-op-match
    M-op Match: (EMIT[SKPE 2 1 1] 7 1 5)
   Attempting Transformations
    Feasible[1]: (-> (NEQ ZACC 0) (+ ZPC 1))
     Transform: (-> (NEQ ZACC 0) $$1:ZMP) => (-> (NEQ ZACC 0) (+ ZPC 1))
      Transform: (NEQ ZACC 0) => (NEQ ZACC 0)
       Transform: ZACC => ZACC
       Transform: 0 => 0
      Transform: $$1:ZMP => (+ ZPC 1)
       Depth Limit Reached
       [fail on $$1:ZMP ]
      [fail on $$1:ZMP ]
      Depth Limit Reached
      [fail on (-> (NEQ ZACC 0) $$1:ZMP) ]
     [fail on (-> (NEQ ZACC 0) $$1:ZMP) ]
    [fail on (-> (NEQ ZACC 0) $$1:ZMP) ]
    Feasible[2]: (-> (EQL $1:Z -1) (+ ZPC 1))
     Transform: (-> (NEQ ZACC 0) $$1:ZMP) => (-> (EQL $1:Z -1) (+ ZPC 1))
      Transform: (NEQ ZACC 0) => (EQL $1:Z -1)
       Depth Limit Reached
       [fail on (NEQ ZACC 0) ]
      [fail on (NEQ ZACC 0) ]
      Depth Limit Reached
      [fail on (-> (NEQ ZACC 0) $$1:ZMP) ]
     [fail on (-> (NEQ ZACC 0) $$1:ZMP) ]
    [fail on (-> (NEQ ZACC 0) $$1:ZMP) ]
   Breadth Limit Reached(6)
   Search: (← ZACC 1)
   Attempting M-op-match
   M-op Match: (EMIT[SET1A 2 1 1] 7 0 30)
   Search: (LABEL $$1:ZMP)
  Applying Skip-Decomposition
   Search: (← ZACC 1)
   Attempting M-op-match
   M-op Match: (EMIT[SET1A 2 1 1] 7 0 30)
   Simplifying (NOT (EQL ZACC 0)) to (NEQ ZACC 0)
   Search: (-> (NEQ ZACC 0) (+ ZPC 1))
   Attempting M-op-match
   M-op Match: (EMIT[SKPNE 2 1 1] 7 1 2)
  Attempting Transformations
   Feasible[1]: (-> (EQL $1:Z -1) (+ ZPC 1))
   Feasible[2]: (-> (EQL ZACC 0) (+ ZPC 1))
   Feasible[3]: (-> (NEQ ZACC 0) (+ ZPC 1))
-----------------------
Nodes Examined: 15
Est. Seconds: .686@-1
Result Sequence(s):
<<<<<<<<<<<<<<<<<<<<<<<<
        SKPE
        JMP     $$1:ZMP
        SET1A
$$1:
-----------or-----------
        SKPNE
        SET1A
>>>>>>>>>>>>>>>>>>>>>>>>
Best Sequence is:
        SKPNE
        SET1A
```

```
-----------------------

 Search: (← 7ACC 0)
 Attempting M-op-match
 M-op Match: (; (ALLOC $$1) (EMIT[DCA 1 1 1] 3 $$1))
 M-op Match: (EMIT[CLRA 3 1 1] 7 0 20)
-----------------------
Nodes Examined: 1.
Est. Seconds: .493@-2
Result Sequence(s):
<<<<<<<<<<<<<<<<<<<<<<<<
       [Alloc $$1]
       DCA     $$1
-----------or-----------
       CLRA
>>>>>>>>>>>>>>>>>>>>>>>>
Best Sequence is:
       CLRA
-----------------------

 Search: (← 7MP 7ACC)
 Attempting M-op-match
 M-op Match: (; (ALLOC 7ACC) (EMIT[DCA 1 1 1] 3 7MP))
-----------------------
Nodes Examined: 1
Est. Seconds: .357@-2
Result Sequence(s):
       [Alloc 7ACC]
       DCA     7MP
Best Sequence is:
       [Alloc 7ACC]
       DCA     7MP
-----------------------

 Search: (← $$1:7MP (+ $$1:7MP 1))
 Attempting M-op-match
 M-op Match: (; (EMIT[ISZ 1 1 1] 2 $$1:7MP) (EMIT[NOP 3 1 1] 7 0 0))
-----------------------
Nodes Examined: 1
Est. Seconds: .531@-2
Result Sequence(s):
       ISZ     $$1:7MP
       NOP
Best Sequence is:
       ISZ     $$1:7MP
       NOP
-----------------------

{PDP-11 examples}

 Search: (← $$1:7M (AND $$2:7M $$3:7M))
 Attempting M-op-match
 Attempting Decompositions
 Attempting Transformations
  Feasible[1]: (← $1:DST (AND $1:DST (NOT $2:SRC)))
   Transform: (← $$1:7M (AND $$2:7M $$3:7M)) => (← $1:DST (AND $1:DST (NOT $2:SRC)))
    Transform: $$1:7M => $1:DST
    Transform: (AND $$2:7M $$3:7M) => (AND $1:DST (NOT $2:SRC))
```

```
Transform: $$2:7M => $1:DST
 Applying Fetch Decomposition to: $$2:7M using: $$1:ZM
  Search: (← $$1:7M $$2:7M)
  Attempting M-op-match
  M-op Match: (EMIT[MOV 2 1 1] 1 $$2:ZM $$1:ZM)
Transform: $$3:7M => (NOT $2:SRC)
 Applying $1 :: (NOT (NOT $1))  to: $$3:ZM
  Transform: (NOT (NOT $$3:7M)) => (NOT $2:SRC)
   Transform: (NOT $$3:7M) => $2:SRC
    Applying Fetch Decomposition to: (NOT $$3:7M) using: $$4:ZR
     Search: (← $$4:7R (NOT $$3:7M))
     Attempting M-op-match
     Attempting Decompositions
     Attempting Transformations
      Feasible[1]: (← $1:DST (NOT $1:DST))
       Transform: (← $$4:ZR (NOT $$3:ZM)) => (← $1:DST (NOT $1:DST))
        Transform: $$4:7R => $1:DST
        Transform: (NOT $$3:7M) => (NOT $1:DST)
         Transform: $$3:7M => $1:DST
          Applying Fetch Decomposition to: $$3:ZM using: $$4:ZR
           Search: (← $$4:7R $$3:7M)
           Attempting M-op-match
           M-op Match: (EMIT[MOV 2 1 1] 1 $$3:ZM $$4:ZR)
     Breadth Limit Reached(3)
Feasible[2]: (← $1:DST (NOT $1:DST))
 Transform: (← $$1:7M (AND $$2:7M $$3:7M)) => (← $1:DST (NOT $1:DST))
  Transform: $$1:7M => $1:DST
  Transform: (AND $$2:7M $$3:7M) => (NOT $1:DST)
   Applying (AND $1 $2) :: (NOT (OR (NOT $1) (NOT $2)))  to: (AND $$2:7M $$3:7M)
    Transform: (NOT (OR (NOT $$2:7M) (NOT $$3:7M))) => (NOT $1:DST)
     Transform: (OR (NOT $$2:7M) (NOT $$3:7M)) => $1:DST
      Applying Fetch Decomposition to: (OR (NOT $$2:7M) (NOT $$3:7M)) using: $$1:ZM
       Search: (← $$1:7M (OR (NOT $$2:7M) (NOT $$3:7M)))
       Attempting M-op-match
       Attempting Decompositions
       Attempting Transformations
        Feasible[1]: (← $1:DST (OR $1:DST $2:SRC))
         Transform: (← $$1:7M (OR (NOT $$2:7M) (NOT $$3:7M))) => (← $1:DST (OR $1:DST $2:SRC))
          Transform: $$1:7M => $1:DST
          Transform: (OR (NOT $$2:7M) (NOT $$3:7M)) => (OR $1:DST $2:SRC)
           Transform: (NOT $$2:7M) => $1:DST
            Applying Fetch Decomposition to: (NOT $$2:7M) using: $$1:ZM
             Search: (← $$1:7M (NOT $$2:7M))
             Attempting M-op-match
             Depth Limit Reached
             [fail on (← $$1:7M (NOT $$2:7M)) ]
            [fail on (NOT $$2:7M) ]
           [fail on (NOT $$2:7M) ]
           [fail on (OR (NOT $$2:7M) (NOT $$3:7M)) ]
          [fail on (OR (NOT $$2:7M) (NOT $$3:7M)) ]
          [fail on (← $$1:7M (OR (NOT $$2:7M) (NOT $$3:7M))) ]
         [fail on (← $$1:7M (OR (NOT $$2:7M) (NOT $$3:7M))) ]
        [fail on (← $$1:7M (OR (NOT $$2:7M) (NOT $$3:7M))) ]
       Breadth Limit Reached(5)
       [fail on (← $$1:7M (OR (NOT $$2:7M) (NOT $$3:7M))) ]
      [fail on (OR (NOT $$2:7M) (NOT $$3:7M)) ]
     [fail on (OR (NOT $$2:7M) (NOT $$3:7M)) ]
     [fail on (NOT (OR (NOT $$2:7M) (NOT $$3:7M))) ]
    [fail on (NOT (OR (NOT $$2:7M) (NOT $$3:7M))) ]
```

```
Applying $1 :: (NOT (NOT $1))  to: (AND $$2:7M $$3:7M)
 Transform: (NOT (NOT (AND $$2:7M $$3:7M))) => (NOT $1:DST)
  Transform: (NOT (AND $$2:7M $$3:7M)) => $1:DST
   Applying Fetch Decomposition to: (NOT (AND $$2:7M $$3:7M)) using: $$1:ZM
    Search: (<- $$1:7M (NOT (AND $$2:7M $$3:7M)))
    Attempting M-op-match
    Attempting Decompositions
    Attempting Transformations
     Feasible[1]: (<- $1:DST (NOT $1:DST))
      Transform: (<- $$1:7M (NOT (AND $$2:7M $$3:7M))) => (<- $1:DST (NOT $1:DST))
       Transform: $$1:7M => $1:DST
       Transform: (NOT (AND $$2:7M $$3:7M)) => (NOT $1:DST)
        Transform: (AND $$2:7M $$3:7M) => $1:DST
         Applying Fetch Decomposition to: (AND $$2:7M $$3:7M) using: $$1:ZM
          Search: (<- $$1:7M (AND $$2:7M $$3:7M))
          Attempting M-op-match
          Depth Limit Reached
          [fail on (<- $$1:7M (AND $$2:7M $$3:7M)) ]
         [fail on (AND $$2:7M $$3:7M) ]
        [fail on (AND $$2:7M $$3:7M) ]
        [fail on (NOT (AND $$2:7M $$3:7M)) ]
       [fail on (NOT (AND $$2:7M $$3:7M)) ]
       [fail on (<- $$1:ZM (NOT (AND $$2:7M $$3:7M))) ]
      [fail on (<- $$1:7M (NOT (AND $$2:7M $$3:7M))) ]
     [fail on (<- $$1:7M (NOT (AND $$2:7M $$3:7M))) ]
     Broadth Limit Reached(5)
     [fail on (<- $$1:7M (NOT (AND $$2:7M $$3:7M))) ]
    [fail on (NOT (AND $$2:7M $$3:7M)) ]
    [fail on (NOT (AND $$2:7M $$3:7M)) ]
    [fail on (NOT (NOT (AND $$2:7M $$3:7M))) ]
   [fail on (NOT (NOT (AND $$2:7M $$3:7M))) ]
  [fail on (AND $$2:7M $$3:7M) ]
 [fail on (AND $$2:7M $$3:7M) ]
 [fail on (<- $$1:7M (AND $$2:7M $$3:7M)) ]
[fail on (<- $$1:7M (AND $$2:7M $$3:7M)) ]
[fail on (<- $$1:7M (AND $$2:7M $$3:7M)) ]
Feasible[3]: (<- $1:DST $2:SRC)
 Transform: (<- $$1:7M (AND $$2:7M $$3:7M)) => (<- $1:DST $2:SRC)
  Transform: $$1:7M => $1:DST
  Transform: (AND $$2:7M $$3:7M) => $2:SRC
   Applying Fetch Decomposition to: (AND $$2:7M $$3:7M) using: $$5:ZR
    Search: (<- $$5:7R (AND $$2:7M $$3:7M))
    Attempting M-op-match
    Attempting Decompositions
    Attempting Transformations
     Feasible[1]: (<- $1:DST (AND $1:DST (NOT $2:SRC)))
      Transform: (<- $$5:7R (AND $$2:7M $$3:7M)) => (<- $1:DST (AND $1:DST (NOT $2:SRC)))
       Transform: $$5:7R => $1:DST
       Transform: (AND $$2:7M $$3:7M) => (AND $1:DST (NOT $2:SRC))
        Transform: $$2:7M => $1:DST
         Applying Fetch Decomposition to: $$2:7M using: $$5:ZR
          Search: (<- $$5:7R $$2:7M)
          Attempting M-op-match
          M-op Match: (EMIT[MOV 2 1 1] 1 $$2:7M $$5:ZR)
        Transform: $$3:7M => (NOT $2:SRC)
         Applying $1 :: (NOT (NOT $1))  to: $$3:ZM
          Transform: (NOT (NOT $$3:7M)) => (NOT $2:SRC)
           Transform: (NOT $$3:7M) => $2:SRC
            Applying Fetch Decomposition to: (NOT $$3:7M) using: $$6:ZR
```

```
                Search: (← $$6:7R (NOT $$3:7M))
                Attempting M-op-match
                Depth Limit Reached
                [fail on (← $$6:7R (NOT $$3:7M)) ]
               [fail on (NOT $$3:7M) ]
              [fail on (NOT $$3:7M) ]
              [fail on (NOT (NOT $$3:7M)) ]
             [fail on (NOT (NOT $$3:7M)) ]
            [fail on $$3:7M ]
           [fail on $$3:7M ]
           [fail on (AND $$2:7M $$3:7M) ]
          [fail on (AND $$2:7M $$3:7M) ]
          [fail on (← $$5:7R (AND $$2:7M $$3:7M)) ]
         [fail on (← $$5:7R (AND $$2:7M $$3:7M)) ]
        [fail on (← $$5:7R (AND $$2:7M $$3:7M)) ]
       Breadth Limit Reached(10)
       [fail on (← $$5:7R (AND $$2:7M $$3:7M)) ]
      [fail on (AND $$2:7M $$3:7M) ]
     [fail on (AND $$2:7M $$3:7M) ]
     [fail on (← $$1:7M (AND $$2:7M $$3:7M)) ]
    [fail on (← $$1:7M (AND $$2:7M $$3:7M)) ]
   [fail on (← $$1:7M (AND $$2:7M $$3:7M)) ]
  Broadth Limit Reached(45)
-------------------------
Nodes Examined: 46
Est. Seconds:  .227
Result Sequence(s):
        [Alloc $$1:7M]
        MOV      $$2:7M $$1:7M
        [Alloc $$4:7R]
        [Alloc $$4:7R]
        MOV      $$3:7M $$4:7R
        COM      $$4:7R
        BIC      $$4:7R $$1:7M
Best Sequence is:
        [Alloc $$1:7M]
        MOV      $$2:7M $$1:7M
        [Alloc $$4:7R]
        [Alloc $$4:7R]
        MOV      $$3:7M $$4:7R
        COM      $$4:7R
        BIC      $$4:7R $$1:7M
-------------------------

Search: (-> (NEQ $$1:7M $$2:7M) $$3)
Attempting M-op-match
Attempting Decompositions
Attempting Transformations
 Feasible[1]: (← $1:DST (NOT $1:DST))
 Feasible[2]: (-> (NOT 7Z) $1:ADR)
  Transform: (-> (NEQ $$1:7M $$2:7M) $$3) => (-> (NOT 7Z) $1:ADR)
   Transform: (NEQ $$1:7M $$2:7M) => (NOT 7Z)
    Applying (NEQ $1 $2) = (NOT (EQL $1 $2))  to: (NEQ $$1:7M $$2:7M)
     Transform: (NOT (EQL $$1:7M $$2:7M)) => (NOT 7Z)
      Transform: (EQL $$1:7M $$2:7M) => 7Z
       Applying Fetch Decomposition to: (EQL $$1:7M $$2:7M) using: $$3:7Z
        Search: (← $$3:7Z (EQL $$1:7M $$2:7M))
        Attempting M-op-match
```

```
      M-op Match: (EMIT[CMP 2 1 1] 2 $$2:7M $$1:7M)
   Transform: $$3 => $1:ADR
Feasible[3]: (-> (NOT 7N) $1:ADR)
 Transform: (-> (NEQ $$1:7M $$2:7M) $$3) => (-> (NOT 7N) $1:ADR)
 Transform: (NEQ $$1:7M $$2:7M) => (NOT 7N)
  Applying (NEQ $1 $2) :: (NOT (EQL $1 $2))  to: (NEQ $$1:7M $$2:7M)
   Transform: (NOT (EQL $$1:7M $$2:7M)) => (NOT 7N)
    Transform: (EQL $$1:7M $$2:7M) => 7N
     Applying Fetch Decomposition to: (EQL $$1:7M $$2:7M) using: $$4:7N
      Search: (← $$4:7N (EQL $$1:7M $$2:7M))
      Attempting M-op-match
      Attempting Decompositions
      Attempting Transformations
       Feasible[1]: (← 7Z (EQL $1:DST 0))
       Attempting Store-Decomposition using: $$5:7Z
        Search: (← $$4:7N $$5:7Z)
        Attempting M-op-match
        Depth Limit Reached
        [fail on (← $$4:7N $$5:7Z) ]
       [fail on (← $$4:7N (EQL $$1:7M $$2:7M)) ]
       Feasible[2]: (← 7Z (EQL $1:DST $2:SRC))
       Attempting Store-Decomposition using: $$6:7Z
        Search: (← $$4:7N $$6:7Z)
        Attempting M-op-match
        Depth Limit Reached
        [fail on (← $$4:7N $$6:7Z) ]
       [fail on (← $$4:7N (EQL $$1:7M $$2:7M)) ]
       Feasible[3]: (← 7Z (EQL (OR $1:DST $2:SRC) 0))
       Attempting Store-Decomposition using: $$7:7Z
        Search: (← $$4:7N $$7:7Z)
        Attempting M-op-match
        Depth Limit Reached
        [fail on (← $$4:7N $$7:7Z) ]
       [fail on (← $$4:7N (EQL $$1:7M $$2:7M)) ]
       Feasible[4]: (← $1:DST (NOT $1:DST))
       Attempting Store-Decomposition using: $$8:7R
        Search: (← $$4:7N $$8:7R)
        Attempting M-op-match
        Depth Limit Reached
        [fail on (← $$4:7N $$8:7R) ]
       [fail on (← $$4:7N (EQL $$1:7M $$2:7M)) ]
       Feasible[5]: (← $1:DST $2:SRC)
       Attempting Store-Decomposition using: $$9:7R
        Search: (← $$4:7N $$9:7R)
        Attempting M-op-match
        Depth Limit Reached
        [fail on (← $$4:7N $$9:7R) ]
       [fail on (← $$4:7N (EQL $$1:7M $$2:7M)) ]
      Breadth Limit Reached(10)
      [fail on (← $$4:7N (EQL $$1:7M $$2:7M)) ]
     [fail on (EQL $$1:7M $$2:7M) ]
    [fail on (EQL $$1:7M $$2:7M) ]
    [fail on (NOT (EQL $$1:7M $$2:7M)) ]
  [fail on (NOT (EQL $$1:7M $$2:7M)) ]
 Applying $1 :: (NOT (NOT $1))  to: (NEQ $$1:7M $$2:7M)
  Transform: (NOT (NOT (NEQ $$1:7M $$2:7M))) => (NOT 7N)
   Transform: (NOT (NEQ $$1:7M $$2:7M)) => 7N
    Applying Fetch Decomposition to: (NOT (NEQ $$1:7M $$2:7M)) using: $$10:7N
     Search: (← $$10:7N (NOT (NEQ $$1:7M $$2:7M)))
```

```
      Attempting M-op-match
      Attempting Decompositions
      Attempting Transformations
       Feasible[1]: (← $1:DST (NOT $1:DST))
       Attempting Store-Decomposition using: $$11:ZR
        Search: (← $$10:7N $$11:7R)
        Attempting M-op-match
        Depth Limit Reached
        [fail on (← $$10:7N $$11:7R) ]
       [fail on (← $$10:7N (NOT (NEQ $$1:7M $$2:7M))) ]
       Feasible[2]: (← $1:DST $2:SRC)
       Attempting Store-Decomposition using: $$12:ZR
        Search: (← $$10:7N $$12:7R)
        Attempting M-op-match
        Depth Limit Reached
        [fail on (← $$10:7N $$12:7R) ]
       [fail on (← $$10:7N (NOT (NEQ $$1:7M $$2:7M))) ]
      No more feasible M-ops
      [fail on (← $$10:7N (NOT (NEQ $$1:7M $$2:7M))) ]
     [fail on (NOT (NEQ $$1:7M $$2:7M)) ]
    [fail on (NOT (NEQ $$1:7M $$2:7M)) ]
    [fail on (NOT (NOT (NEQ $$1:7M $$2:7M))) ]
   [fail on (NOT (NOT (NEQ $$1:7M $$2:7M))) ]
  [fail on (NEQ $$1:7M $$2:7M) ]
 [fail on (NEQ $$1:7M $$2:7M) ]
 [fail on (-> (NEQ $$1:7M $$2:7M) $$3) ]
[fail on (-> (NEQ $$1:7M $$2:7M) $$3) ]
[fail on (-> (NEQ $$1:7M $$2:7M) $$3) ]
Feasible[4]: (← 7Z (EQL $1:DST 0))
Feasible[5]: (← 7Z (EQL $1:DST $2:SRC))
Feasible[6]: (← 7Z (EQL (OR $1:DST $2:SRC) 0))
Feasible[7]: (; (← $1:DST $2:SRC) (← 7N (LSS $2:SRC 0)) (← 7Z (EQL $2:SRC 0)))
Feasible[8]: (← $1:DST $2:SRC)
Feasible[9]: (← 7PC $1:ADR)
Feasible[10]: (-> 7Z $1:ADR)
 Transform: (-> (NEQ $$1:7M $$2:7M) $$3) => (-> 7Z $1:ADR)
  Transform: (NEQ $$1:7M $$2:7M) => 7Z
   Applying Fetch Decomposition to: (NEQ $$1:7M $$2:7M) using: $$13:7Z
    Search: (← $$13:7Z (NEQ $$1:7M $$2:7M))
    Attempting M-op-match
    Attempting Decompositions
    Attempting Transformations
     Feasible[1]: (← 7Z (EQL $1:DST 0))
      Transform: (← $$13:7Z (NEQ $$1:7M $$2:7M)) => (← 7Z (EQL $1:DST 0))
       Transform: $$13:7Z => 7Z
       Transform: (NEQ $$1:7M $$2:7M) => (EQL $1:DST 0)
        [fail on (NEQ $$1:7M $$2:7M) ]
       [fail on (NEQ $$1:7M $$2:7M) ]
       [fail on (← $$13:7Z (NEQ $$1:7M $$2:7M)) ]
      [fail on (← $$13:7Z (NEQ $$1:7M $$2:7M)) ]
     [fail on (← $$13:7Z (NEQ $$1:7M $$2:7M)) ]
     Feasible[2]: (← 7Z (EQL $1:DST $2:SRC))
      Transform: (← $$13:7Z (NEQ $$1:7M $$2:7M)) => (← 7Z (EQL $1:DST $2:SRC))
       Transform: $$13:7Z => 7Z
       Transform: (NEQ $$1:7M $$2:7M) => (EQL $1:DST $2:SRC)
        [fail on (NEQ $$1:7M $$2:7M) ]
       [fail on (NEQ $$1:7M $$2:7M) ]
       [fail on (← $$13:7Z (NEQ $$1:7M $$2:7M)) ]
      [fail on (← $$13:7Z (NEQ $$1:7M $$2:7M)) ]
```

```
    [fail on (← $$13:7Z (NEQ $$1:7M $$2:7M)) ]
    Feasible[3]: (← 7Z (EQL (OR $1:DST $2:SRC) 0))
     Transform: (← $$13:7Z (NEQ $$1:7M $$2:7M)) => (← 7Z (EQL (OR $1:DST $2:SRC) 0))
      Transform: $$13:7Z => 7Z
      Transform: (NEQ $$1:7M $$2:7M) => (EQL (OR $1:DST $2:SRC) 0)
       [fail on (NEQ $$1:7M $$2:7M) ]
      [fail on (NEQ $$1:7M $$2:7M) ]
      [fail on (← $$13:7Z (NEQ $$1:7M $$2:7M)) ]
     [fail on (← $$13:7Z (NEQ $$1:7M $$2:7M)) ]
    [fail on (← $$13:7Z (NEQ $$1:7M $$2:7M)) ]
    Feasible[4]: (← $1:DST (NOT $1:DST))
    Attempting Store-Decomposition using: $$14:7R
     Search: (← $$13:7Z $$14:7R)
     Attempting M-op-match
     Depth Limit Reached
     [fail on (← $$13:7Z $$14:7R) ]
    [fail on (← $$13:7Z (NEQ $$1:7M $$2:7M)) ]
    Feasible[5]: (← $1:DST $2:SRC)
    Attempting Store-Decomposition using: $$15:7R
     Search: (← $$13:7Z $$15:7R)
     Attempting M-op-match
     Depth Limit Reached
     [fail on (← $$13:7Z $$15:7R) ]
    [fail on (← $$13:7Z (NEQ $$1:7M $$2:7M)) ]
   No more feasible M-ops
   [fail on (← $$13:7Z (NEQ $$1:7M $$2:7M)) ]
  [fail on (NEQ $$1:7M $$2:7M) ]
 [fail on (NEQ $$1:7M $$2:7M) ]
 [fail on (-> (NEQ $$1:7M $$2:7M) $$3) ]
[fail on (-> (NEQ $$1:7M $$2:7M) $$3) ]
[fail on (-> (NEQ $$1:7M $$2:7M) $$3) ]
Feasible[11]: (-> 7N $1:ADR)
 Transform: (-> (NEQ $$1:7M $$2:7M) $$3) => (-> 7N $1:ADR)
  Transform: (NEQ $$1:7M $$2:7M) => 7N
   Applying Fetch Decomposition to: (NEQ $$1:7M $$2:7M) using: $$16:7N
    Search: (← $$16:7N (NEQ $$1:7M $$2:7M))
    Attempting M-op-match
    Attempting Decompositions
    Attempting Transformations
     Feasible[1]: (← $1:DST (NOT $1:DST))
     Attempting Store-Decomposition using: $$17:7R
      Search: (← $$16:7N $$17:7R)
      Attempting M-op-match
      Depth Limit Reached
      [fail on (← $$16:7N $$17:7R) ]
     [fail on (← $$16:7N (NEQ $$1:7M $$2:7M)) ]
     Feasible[2]: (← 7Z (EQL $1:DST 0))
     Attempting Store-Decomposition using: $$18:7Z
      Search: (← $$16:7N $$18:7Z)
      Attempting M-op-match
      Depth Limit Reached
      [fail on (← $$16:7N $$18:7Z) ]
     [fail on (← $$16:7N (NEQ $$1:7M $$2:7M)) ]
     Feasible[3]: (← 7Z (EQL $1:DST $2:SRC))
     Attempting Store-Decomposition using: $$19:7Z
      Search: (← $$16:7N $$19:7Z)
      Attempting M-op-match
      Depth Limit Reached
      [fail on (← $$16:7N $$19:7Z) ]
```

```
      [fail on (← $$16:7N (NEQ $$1:7M $$2:7M)) ]
      Feasible[4]: (← 7Z (EQL (OR $1:DST $2:SRC) 0))
      Attempting Store-Decomposition using: $$20:ZZ
       Search: (← $$16:7N $$20:7Z)
       Attempting M-op-match
       Depth Limit Reached
       [fail on (← $$16:7N $$20:7Z) ]
      [fail on (← $$16:7N (NEQ $$1:7M $$2:7M)) ]
      Feasible[5]: (← $1:DST $2:SRC)
      Attempting Store-Decomposition using: $$21:ZR
       Search: (← $$16:7N $$21:7R)
       Attempting M-op-match
       Depth Limit Reached
       [fail on (← $$16:7N $$21:7R) ]
      [fail on (← $$16:7N (NEQ $$1:7M $$2:7M)) ]
     No more feasible M-ops
     [fail on (← $$16:7N (NEQ $$1:7M $$2:7M)) ]
    [fail on (NEQ $$1:7M $$2:7M) ]
   [fail on (NEQ $$1:7M $$2:7M) ]
   [fail on (-> (NEQ $$1:7M $$2:7M) $$3) ]
  [fail on (-> (NEQ $$1:7M $$2:7M) $$3) ]
 [fail on (-> (NEQ $$1:7M $$2:7M) $$3) ]
No more feasible M-ops
-----------------------
Nodes Examined: 62
Est. Seconds:  .288
Result Sequence(s):
      [Alloc $$3:7Z]
      CMP     $$2:7M $$1:7M
      BNE     $$3
Best Sequence is:
      [Alloc $$3:7Z]
      CMP     $$2:7M $$1:7M
      BNE     $$3
-----------------------

Search: (-> (LEQ $$1:7M $$2:7M) $$3)
Attempting M-op-match
Attempting Decompositions
Attempting Transformations
 Feasible[1]: (← $1:DST (OR $1:DST $2:SRC))
 Feasible[2]: (-> (OR 7Z 7N) $1:ADR)
  Transform: (-> (LEQ $$1:7M $$2:7M) $$3) => (-> (OR ZZ ZN) $1:ADR)
   Transform: (LEQ $$1:7M $$2:7M) => (OR ZZ ZN)
    Applying (LEQ $1 $2) = (OR (EQL $1 $2) (LSS $1 $2))  to: (LEQ $$1:ZM $$2:ZM)
     Transform: (OR (EQL $$1:7M $$2:7M) (LSS $$1:7M $$2:7M)) => (OR ZZ ZN)
      Transform: (EQL $$1:7M $$2:7M) => ZZ
       Applying Fetch Decomposition to: (EQL $$1:7M $$2:7M) using: $$3:ZZ
        Search: (← $$3:7Z (EQL $$1:7M $$2:7M))
        Attempting M-op-match
        M-op Match: (EMIT[CMP 2 1 1] 2 $$1:7M $$2:7M)
      Transform: (LSS $$1:7M $$2:7M) => 7N
       Applying Fetch Decomposition to: (LSS $$1:7M $$2:7M) using: $$4:ZN
        Search: (← $$4:7N (LSS $$1:7M $$2:7M))
        Attempting M-op-match
        M-op Match: (EMIT[CMP 2 1 1] 2 $$1:7M $$2:7M)
   Transform: $$3 => $1:ADR
 Feasible[3]: (← $1:DST (NOT $1:DST))
```

```
Feasible[4]: (-> (NOT 7Z) $1:ADR)
 Transform: (-> (LEQ $$1:7M $$2:7M) $$3) => (-> (NOT 7Z) $1:ADR)
  Transform: (LEQ $$1:7M $$2:7M) => (NOT 7Z)
   Applying (LEQ $1 $2) :: (NOT (GTR $1 $2))  to: (LEQ $$1:7M $$2:7M)
    Transform: (NOT &) => (NOT 7Z)
     Transform: & => 7Z
      Applying Fetch Decomposition to: & using: $$5:7Z
       Search: (<- $$5:7Z (GTR $$1:7M $$2:7M))
       Attempting M-op-match
       Attempting Decompositions
       Attempting Transformations
        Feasible[1]: (<- $1:DST (NOT $1:DST))
        Attempting Store-Decomposition using: $$6:7R
         Search: (<- $$5:7Z $$6:7R)
         Attempting M-op-match
         Depth Limit Reached
         [fail on (<- $$5:7Z $$6:7R) ]
        [fail on (<- $$5:7Z (GTR $$1:7M $$2:7M)) ]
        Feasible[2]: (<- $1:DST (AND $1:DST (NOT $2:SRC)))
        Attempting Store-Decomposition using: $$7:7R
         Search: (<- $$5:7Z $$7:7R)
         Attempting M-op-match
         Depth Limit Reached
         [fail on (<- $$5:7Z $$7:7R) ]
        [fail on (<- $$5:7Z (GTR $$1:7M $$2:7M)) ]
        Feasible[3]: (<- $1:DST $2:SRC)
        Attempting Store-Decomposition using: $$8:7R
         Search: (<- $$5:7Z $$8:7R)
         Attempting M-op-match
         Depth Limit Reached
         [fail on (<- $$5:7Z $$8:7R) ]
        [fail on (<- $$5:7Z (GTR $$1:7M $$2:7M)) ]
       Breadth Limit Reached(6)
       [fail on (<- $$5:7Z (GTR $$1:7M $$2:7M)) ]
      [fail on (GTR $$1:7M $$2:7M) ]
     [fail on (GTR $$1:7M $$2:7M) ]
     [fail on (NOT (GTR $$1:7M $$2:7M)) ]
    [fail on (NOT (GTR $$1:7M $$2:7M)) ]
   Applying $1 :: (NOT (NOT $1))  to: (LEQ $$1:7M $$2:7M)
    Transform: (NOT (NOT (LEQ $$1:7M $$2:7M))) => (NOT 7Z)
     Transform: (NOT (LEQ $$1:7M $$2:7M)) => 7Z
      Applying Fetch Decomposition to: (NOT (LEQ $$1:7M $$2:7M)) using: $$9:7Z
       Search: (<- $$9:7Z (NOT (LEQ $$1:7M $$2:7M)))
       Attempting M-op-match
       Attempting Decompositions
       Attempting Transformations
        Feasible[1]: (<- $1:DST (NOT $1:DST))
        Attempting Store-Decomposition using: $$10:7R
         Search: (<- $$9:7Z $$10:7R)
         Attempting M-op-match
         Depth Limit Reached
         [fail on (<- $$9:7Z $$10:7R) ]
        [fail on (<- $$9:7Z (NOT (LEQ $$1:7M $$2:7M))) ]
        Feasible[2]: (<- $1:DST $2:SRC)
        Attempting Store-Decomposition using: $$11:7R
         Search: (<- $$9:7Z $$11:7R)
         Attempting M-op-match
         Depth Limit Reached
         [fail on (<- $$9:7Z $$11:7R) ]
```

```
        [fail on (← $$9:7Z (NOT (LEQ $$1:7M $$2:7M))) ]
        No more feasible M-ops
        [fail on (← $$9:7Z (NOT (LEQ $$1:7M $$2:7M))) ]
       [fail on (NOT (LEQ $$1:7M $$2:7M)) ]
      [fail on (NOT (LEQ $$1:7M $$2:7M)) ]
      [fail on (NOT (NOT (LEQ $$1:7M $$2:7M))) ]
     [fail on (NOT (NOT (LEQ $$1:7M $$2:7M))) ]
    [fail on (LEQ $$1:7M $$2:7M) ]
   [fail on (LEQ $$1:7M $$2:7M) ]
   [fail on (-> (LEQ $$1:7M $$2:7M) $$3) ]
  [fail on (-> (LEQ $$1:7M $$2:7M) $$3) ]
 [fail on (-> (LEQ $$1:7M $$2:7M) $$3) ]
Feasible[5]: (-> (NOT 7N) $1:ADR)
 Transform: (-> (LEQ $$1:7M $$2:7M) $$3) => (-> (NOT 7N) $1:ADR)
  Transform: (LEQ $$1:7M $$2:7M) => (NOT 7N)
   Applying (LEQ $1 $2) :: (NOT (GTR $1 $2))  to: (LEQ $$1:7M $$2:7M)
    Transform: (NOT (GTR $$1:7M $$2:7M)) => (NOT 7N)
     Transform: (GTR $$1:7M $$2:7M) => 7N
      Applying Fetch Decomposition to: (GTR $$1:7M $$2:7M) using: $$12:7N
       Search: (← $$12:7N (GTR $$1:7M $$2:7M))
       Attempting M-op-match
       Attempting Decompositions
       Attempting Transformations
        Feasible[1]: (← $1:DST (NOT $1:DST))
        Attempting Store-Decomposition using: $$13:7R
         Search: (← $$12:7N $$13:7R)
         Attempting M-op-match
         Depth Limit Reached
         [fail on (← $$12:7N $$13:7R) ]
        [fail on (← $$12:7N (GTR $$1:7M $$2:7M)) ]
        Feasible[2]: (← $1:DST (AND $1:DST (NOT $2:SRC)))
        Attempting Store-Decomposition using: $$14:7R
         Search: (← $$12:7N $$14:7R)
         Attempting M-op-match
         Depth Limit Reached
         [fail on (← $$12:7N $$14:7R) ]
        [fail on (← $$12:7N (GTR $$1:7M $$2:7M)) ]
        Feasible[3]: (← $1:DST $2:SRC)
        Attempting Store-Decomposition using: $$15:7R
         Search: (← $$12:7N $$15:7R)
         Attempting M-op-match
         Depth Limit Reached
         [fail on (← $$12:7N $$15:7R) ]
        [fail on (← $$12:7N (GTR $$1:7M $$2:7M)) ]
       Breadth Limit Reached(6)
       [fail on (← $$12:7N (GTR $$1:7M $$2:7M)) ]
      [fail on (GTR $$1:7M $$2:7M) ]
     [fail on (GTR $$1:7M $$2:7M) ]
     [fail on (NOT (GTR $$1:7M $$2:7M)) ]
    [fail on (NOT (GTR $$1:7M $$2:7M)) ]
   Applying $1 :: (NOT (NOT $1))  to: (LEQ $$1:7M $$2:7M)
    Transform: (NOT (NOT (LEQ $$1:7M $$2:7M))) => (NOT 7N)
    Transform: (NOT (LEQ $$1:7M $$2:7M)) => 7N
     Applying Fetch Decomposition to: (NOT (LEQ $$1:7M $$2:7M)) using: $$16:7N
      Search: (← $$16:7N (NOT (LEQ $$1:7M $$2:7M)))
      Attempting M-op-match
      Attempting Decompositions
      Attempting Transformations
       Feasible[1]: (← $1:DST (NOT $1:DST))
```

```
        Attempting Store-Decomposition using: $$17:ZR
         Search: (← $$16:7N $$17:7R)
         Attempting M-op-match
         Depth Limit Reached
         [fail on (← $$16:7N $$17:7R) ]
        [fail on (← $$16:7N (NOT (LEQ $$1:7M $$2:7M))) ]
        Feasible[2]: (← $1:DST $2:SRC)
        Attempting Store-Decomposition using: $$18:ZR
         Search: (← $$16:7N $$18:7R)
         Attempting M-op-match
         Depth Limit Reached
         [fail on (← $$16:7N $$18:7R) ]
        [fail on (← $$16:7N (NOT (LEQ $$1:7M $$2:7M))) ]
       No more feasible M-ops
       [fail on (← $$16:7N (NOT (LEQ $$1:7M $$2:7M))) ]
      [fail on (NOT (LEQ $$1:7M $$2:7M)) ]
     [fail on (NOT (LEQ $$1:7M $$2:7M)) ]
     [fail on (NOT (NOT (LEQ $$1:7M $$2:7M))) ]
    [fail on (NOT (NOT (LEQ $$1:7M $$2:7M))) ]
   [fail on (LEQ $$1:7M $$2:7M) ]
  [fail on (LEQ $$1:7M $$2:7M) ]
  [fail on (-> (LEQ $$1:7M $$2:7M) $$3) ]
 [fail on (-> (LEQ $$1:7M $$2:7M) $$3) ]
[fail on (-> (LEQ $$1:7M $$2:7M) $$3) ]
Breadth Limit Reached(49)
-----------------------
Nodes Examined: 50
Est Seconds: .225
Result Sequence(s):
      [Alloc $$3:7Z]
      CMP     $$1:7M $$2:7M
      [Alloc $$4:7N]
      CMP     $$1:7M $$2:7M
      BLE     $$3
Best Sequence is:
      [Alloc $$3:7Z]
      CMP     $$1:7M $$2:7M
      [Alloc $$4:7N]
      CMP     $$1:7M $$2:7M
      BLE     $$3
-----------------------
```

Appendix E: Code Selection Example

This is a trace of a user's interaction with the code generator generator. PDP11.MOP is given as the input file. The SELECT routine, which has five passes as described in chapter 4, creates the PDP11.LOP file. The output after the •Pass n• messages is the actual representation of the LOP table.

```
.run cgg
[14:02:58]

CGG V.41
Input file? pdp11.mop

Reading PDP11.MOP
Ifldz: OPCODE1 OPCODE2 OPCODEB OPCODEJ OPCODER OFFSETB SRCMODE SRCREG
SRCINDEX DSTMODE DSTREG DSTINDEX
SBs: M N Z V C PC SP R
AMs: 78 716 7SP 7PC 7N 7Z 7C 7L 7R 7RB 7M 7MB 7@R 7@RB 7@M 7@MB 7R•C
7R•CB 7@R•C 7@R•CB 7•R 7•RB 7-R 7-RB
OCs: SRC SRCB DST DSTB SRCR DSTR ADR
FMTs: 1 2 3 4 5 6 7 8 9 10 11 12 13 14
M-ops: CLR COM INC DEC NEG TST ASR ASL MOV ADD SUB CMP BIT BIC BIS
JSR RTS JMP BR BEQ BNE BLT BGE BLE BGT

Indexing MOP
Setup time:  1.94
{CGG V.41 output for PDP11.LOP}

•Pass 0•
{I-flds} [
(OPCODE1 0 10 0 0)
(OPCODE2 0 4 0 0)
(OPCODEB 0 8 0 0)
(OPCODEJ 0 7 0 0)
(OPCODER 0 13 0 0)
(OFFSETB 8 8 0 D)
(SRCMODE 4 3 0 D)
(SRCREG 7 3 0 D)
(SRCINDEX 0 16 1 D)
(DSTMODE 10 3 0 C)
(DSTREG 13 3 0 D)
(DSTINDEX 0 16 2 D) ]

{SBs} [
(M 65536 8 M)
(N 1 1 C)
(Z 1 1 C)
(V 1 1 C)
(C 1 1 C)
(PC 1 16 P)
(SP 1 16 R)
(R 6 16 G) ]

{AMs} [
78:    $1:=8
716:   $1:=16
7SP:   (<> SP 0 0 16)
7PC:   (<> PC 0 0 16)
```

```
7N:     (<> N 0 0 1)
7Z:     (<> Z 0 0 1)
7C:     (<> C 0 0 1)
7L:     (<> M (+ 7PC (1 $1:=8 1)) 0 16)
7R:     (<> R $1:=3 0 16)
7RB:    (<> R $1:=3 8 8)
7M:     (<> M $1:=16 0 16)
7MB:    (<> M $1:=16 0 8)
7@R:    (<> M (<> R $1:=3 0 16) 0 16)
7@RB:   (<> M (<> R $1:=3 0 16) 8 8)
7@M:    (<> M (<> M $1:=16 0 16) 0 16)
7@MB:   (<> M (<> M $1:=16 0 16) 0 8)
7R•C:   (<> M (+ (<> R $1:=3 0 16) $2:=16) 0 16)
7R•CB:  (<> M (+ (<> R $1:=3 0 16) $2:=16) 0 8)
7@R•C:  (<> M (<> M (+ (<> R $1:=3 0 16) $2:=16) 0 16) 0 16)
7@R•CB:         (<> M (<> M (+ (<> R $1:=3 0 16) $2:=16) 0 16) 0 8)
7+R:    (<> M (← 7R (+ 7R 2)) 0 16)
7+RB:   (<> M (← 7R (+ 7R 1)) 0 8)
7-R:    (<> M (← 7R (- 7R 2)) 0 16)
7-RB:   (<> M (← 7R (- 7R 1)) 0 8) ]

{OC#} [

SRC: (
 7.16 :: (EMIT[ 13 0 1] 2 7 $1)
 7R :: (EMIT[ 12 0 0] 0 $1)
 7M :: (EMIT[ 13 1.50 1] 3 7 $1)
 7@R :: (EMIT[ 12 1.50 0] 1 $1)
 7@M :: (EMIT[ 13 2.70 1] 7 7 $1)
 7R•C :: (EMIT[ 13 2.70 1] 6 $1 $2)
 7@R•C :: (EMIT[ 13 3.90 1] 7 $1 $2) )

SRCB: (
 7RB :: (EMIT[ 12 0 0] 0 $1)
 7MB :: (EMIT[ 13 1.50 1] 3 7 $1)
 7@RB :: (EMIT[ 12 1.50 0] 1 $1)
 7@MB :: (EMIT[ 13 2.70 1] 7 7 $1)
 7R•CB :: (EMIT[ 13 2.70 1] 6 $1 $2)
 7@R•CB :: (EMIT[ 13 3.90 1] 7 $1 $2) )

DST: (
 7.16 :: (EMIT[ 10 0 1] 2 7 $1)
 7R :: (EMIT[ 9 0 0] 0 $1)
 7M :: (EMIT[ 10 1.40 1] 3 7 $1)
 7@R :: (EMIT[ 9 1.40 0] 1 $1)
 7@M :: (EMIT[ 10 2.60 1] 7 7 $1)
 7R•C :: (EMIT[ 10 2.60 1] 6 $1 $2)
 7@R•C :: (EMIT[ 10 3.80 1] 7 $1 $2) )

DSTB: (
 7RB :: (EMIT[ 9 0 0] 0 $1)
 7MB :: (EMIT[ 10 1.40 1] 3 7 $1)
 7@RB :: (EMIT[ 9 1.40 0] 1 $1)
 7@MB :: (EMIT[ 10 2.60 1] 7 7 $1)
 7R•CB :: (EMIT[ 10 2.60 1] 6 $1 $2)
 7@R•CB :: (EMIT[ 10 3.80 1] 7 $1 $2) )
```

```
SRCR: (
 7R :: (EMIT[ 11 0 0] $1) )

DSTR: (
 7R :: (EMIT[ 8 0 0] $1) )

ADR: (
 7L :: (EMIT[ 14 0 0] $1) ) ]

{FMTs} [
{FMT 1} ( OPCODE1 DST)
{FMT 2} ( OPCODE2 SRC DST)
{FMT 3} ( OPCODEB ADR)
{FMT 4} ( OPCODEJ SRCR DST)
{FMT 5} ( OPCODER DSTR)
{FMT 6} ( OPCODE1 DSTB)
{FMT 7} ( OPCODE2 SRCB DSTB)
{FMT 8} ( DSTREG)
{FMT 9} ( DSTMODE DSTREG)
{FMT 10} ( DSTMODE DSTREG DSTINDEX)
{FMT 11} ( SRCREG)
{FMT 12} ( SRCMODE SRCREG)
{FMT 13} ( SRCMODE SRCREG SRCINDEX)
{FMT 14} ( OFFSETB)  ]

.Pass 1.
[
{M-op templates}
 (; (← $1:DST 0) (← 7N (LSS 0 0)) (← 7Z (EQL 0 0)))
 :: (EMIT[CLR 1 1 1] 50 $1)
 (; (← $1 DST (NOT $1:DST)) (← 7N (LSS (NOT $1:DST) 0)) (← 7Z (EQL (NOT $1:DST) 0)))
 :: (EMIT[COM 1 1 1] 51 $1)
 (; (← $1:DST (+ $1:DST 1)) (← 7N (LSS (+ $1:DST 1) 0)) (← 7Z (EQL (+ $1:DST 1) 0)))
 :: (EMIT[INC 1 1 1] 52 $1)
 (; (← $1:DST (- $1:DST 1)) (← 7N (LSS (- $1:DST 1) 0)) (← 7Z (EQL (- $1:DST 1) 0)))
 :: (EMIT[DEC 1 1 1] 53 $1)
 (; (← $1:DST (- $1:DST)) (← 7N (GEQ $1:DST 0)) (← 7Z (EQL $1:DST 0)))
 :: (EMIT[NEG 1 1 1] 54 $1)
 (; (← 7N (LSS $1:DST 0)) (← 7Z (EQL $1:DST 0)))
 :: (EMIT[TST 1 1 1] 57 $1)
 (; (← $1:DST (↑ $1:DST -1)) (← 7N (LSS (↑ $1:DST -1) 0)) (← 7Z (EQL (↑ $1:DST -1) 0)))
 :: (EMIT[ASR 1 1 1] 62 $1)
 (; (← $1:DST (↑ $1:DST 1)) (← 7N (LSS (↑ $1:DST 1) 0)) (← 7Z (EQL (↑ $1:DST 1) 0)))
 :: (EMIT[ASL 1 1 1] 63 $1)
 (; (← $1:DST $2:SRC) (← 7N (LSS $2:SRC 0)) (← 7Z (EQL $2:SRC 0)))
 :: (EMIT[MOV 2 1 1] 1 $2 $1)
 (; (← $1:DST (+ $1:DST $2:SRC)) (← 7N (LSS (+ $1:DST $2:SRC) 0)) (← 7Z (EQL (+ $1:DST $2:SRC) 0)))
 :: (EMIT[ADD 2 1 1] 6 $2 $1)
 (; (← $1:DST (- $1:DST $2:SRC)) (← 7N (LSS $1:DST $2:SRC)) (← 7Z (EQL $1:DST $2:SRC)))
 :: (EMIT[SUB 2 1 1] 16 $2 $1)
 (; (← 7N (LSS $2:DST $1:SRC)) (← 7Z (EQL $2:DST $1:SRC)))
 :: (EMIT[CMP 2 1 1] 2 $2 $1)
 (; (← 7N (LSS (OR $1:DST $2:SRC) 0)) (← 7Z (EQL (OR $1:DST $2:SRC) 0)))
 :: (EMIT[BIT 2 1 1] 3 $2 $1)
 (; (← $1:DST (AND $1:DST (NOT $2:SRC))) (← 7N (LSS (AND $1:DST (NOT $2:SRC)) 0))
    (← 7Z (EQL (AND $1:DST (NOT $2:SRC)) 0))) :: (EMIT[BIC 2 1 1] 4 $2 $1)
 (; (← $1:DST (OR $1:DST $2:SRC)) (← 7N (LSS (OR $1:DST $2:SRC) 0)) (← 7Z (EQL (OR $1:DST $2:SRC) 0)))
 :: (EMIT[BIS 2 1 1] 5 $2 $1)
```

```
(CALL) :: (EMIT[JSR 4 1 1] 4 7 0)
(RETURN) :: (EMIT[RTS 5 1 1] 20 7)
(← 7PC $1:ADR) :: (EMIT[JMP 4 1 1] 0 1 $1)
(← 7PC $1:ADR) :: (EMIT[BR 3 1 1] 4 $1)
(-> 7Z $1:ADR) :: (EMIT[BEQ 3 1 1] 14 $1)
(-> (NOT 7Z) $1:ADR) :: (EMIT[BNE 3 1 1] 10 $1)
(-> 7N $1:ADR) :: (EMIT[BLT 3 1 1] 24 $1)
(-> (NOT 7N) $1:ADR) :: (EMIT[BGE 3 1 1] 20 $1)
(-> (OR 7Z 7N) $1:ADR) :: (EMIT[BLE 3 1 1] 34 $1)
(-> (AND (NOT 7Z) (NOT 7N)) $1:ADR) :: (EMIT[BGT 3 1 1] 30 $1)
•Pass 2•
{P-M-op templates}
(← $1:DST 0) :: (EMIT[CLR 1 1 1] 50 $1:DST)
(← $1:DST (NOT $1:DST)) :: (EMIT[COM 1 1 1] 51 $1:DST)
(← $1:DST (+ $1:DST 1)) :: (EMIT[INC 1 1 1] 52 $1:DST)
(← $1:DST (- $1:DST 1)) :: (EMIT[DEC 1 1 1] 53 $1:DST)
(← $1:DST (- $1:DST)) :: (EMIT[NEG 1 1 1] 54 $1:DST)
(← 7N (LSS $1:DST 0)) :: (EMIT[TST 1 1 1] 57 $1:DST)
(← 7Z (EQL $1:DST 0)) :: (EMIT[TST 1 1 1] 57 $1:DST)
(← $1:DST (↑ $1:DST -1)) :: (EMIT[ASR 1 1 1] 62 $1:DST)
(← $1:DST (↑ $1:DST 1)) :: (EMIT[ASL 1 1 1] 63 $1:DST)
(← $1:DST $2:SRC) :: (EMIT[MOV 2 1 1] 1 $2:SRC $1:DST)
(← $1:DST (+ $1:DST $2:SRC)) :: (EMIT[ADD 2 1 1] 6 $2:SRC $1:DST)
(← $1:DST (- $1:DST $2:SRC)) :: (EMIT[SUB 2 1 1] 16 $2:SRC $1:DST)
(← 7N (LSS $2:DST $1:SRC)) :: (EMIT[CMP 2 1 1] 2 $2:DST $1:SRC)
(← 7Z (EQL $2:DST $1:SRC)) :: (EMIT[CMP 2 1 1] 2 $2:DST $1:SRC)
(← 7N (LSS (OR $1:DST $2:SRC) 0)) :: (EMIT[BIT 2 1 1] 3 $2:SRC $1:DST)
(← 7Z (EQL (OR $1:DST $2:SRC) 0)) :: (EMIT[BIT 2 1 1] 3 $2:SRC $1:DST)
(← $1:DST (AND $1:DST (NOT $2:SRC))) :: (EMIT[BIC 2 1 1] 4 $2:SRC $1:DST)
(← $1:DST (OR $1:DST $2:SRC)) :: (EMIT[BIS 2 1 1] 5 $2:SRC $1:DST)
•Pass 3•
{derived templates}
{No entry made for (← $$1:7RB $$2:7MB)}
{No entry made for (← $$1:7MB $$2:7RB)}
•Pass 4•
{No entry made for (← $$40 &DST (• & &))}
{No entry made for (← $$20 &DST (/ & &))}
{No entry made for (← $$20 &DST (•F & &))}
{No entry made for (← $$20 &DST (-F & &))}
{No entry made for (← $$20 &DST (•F & &))}
{No entry made for (← $$20 &DST (/F & &))}
(← $1:DST (AND $1:DST $2:7R))
:: (; (ALLOC $2:7R) (EMIT[COM 1 1 1] 51 $2:7R) (EMIT[BIC 2 1 1] 4 $2:7R $1:DST))
(GOTO $1) :: (EMIT[BR 3 1 1] 4 $1)
(-> (EQL $1:DST $2:SRC) $3)
:: (; (ALLOC $4:7Z) (EMIT[CMP 2 1 1] 2 $1:DST $2:SRC) (EMIT[BEQ 3 1 1] 14 $3))
(-> (NEQ $1:DST $2:SRC) $3)
:: (; (ALLOC $4:7Z) (EMIT[CMP 2 1 1] 2 $1:DST $2:SRC) (EMIT[BNE 3 1 1] 10 $3))
(-> (GTR $1:DST $2:SRC) $3)
:: (; (ALLOC $4:7Z) (EMIT[CMP 2 1 1] 2 $5:DST $6:SRC)
   (ALLOC $7:7N) (EMIT[CMP 2 1 1] 2 $1:DST $2:SRC) (EMIT[BGT 3 1 1] 30 $3))
(-> (LSS $1:DST $2:SRC) $3)
:: (; (ALLOC $4:7N) (EMIT[CMP 2 1 1] 2 $1:DST $2:SRC) (EMIT[BLT 3 1 1] 24 $3))
(-> (GEQ $1:DST $2:SRC) $3)
:: (; (ALLOC $3:7N) (EMIT[CMP 2 1 1] 2 $1:DST $2:SRC) (EMIT[BGE 3 1 1] 20 $3))
(-> (LEQ $1:DST $2:SRC) $3)
:: (; (ALLOC $4:7Z) (EMIT[CMP 2 1 1] 2 $5:DST $6:SRC)
   (ALLOC $7:7N) (EMIT[CMP 2 1 1] 2 $1:DST $2:SRC) (EMIT[BLE 3 1 1] 34 $3))
•Pass 5•
```

```
{fixed templates}
 (WHILE $1 $2) :: (; (LABEL $3) (-> (NOT $1) $4) $2 (GOTO $3) (LABEL $4))
 (IF $1 $2) :: (; (-> (NOT $1) $3) $2 (LABEL $3))
 (IF $1 $2 $3) :: (; (-> $1 $4) $3 (GOTO $5) (LABEL $4) $2 (LABEL $5)) ]

End of SAIL execution
[9.03  ]
```

Appendix F: Code Generator Generator Axioms

```
!Axioms used by the code generator generator!

3 :: (+ 2 1)                2 :: (+ 1 1)
$1 :: (+ 0 $1)
$1 :: (- (- $1))
(- $1) :: (- 0 $1)
(- $1) :: (+ 0 (- $1))
(- $1 $2) :: (+ $1 (- $2))
(+ $1 $2) :: (+ $2 $1)
(* $1 $2) :: (* $2 $1)

$1 :: (NOT (NOT $1))
(AND $1 $2) :: (NOT (OR (NOT $1) (NOT $2)))
(OR $1 $2) :: (NOT (AND (NOT $1) (NOT $2)))

(LSS $1 $2) :: (NOT (GEQ $1 $2))
(LEQ $1 $2) :: (OR (EQL $1 $2) (LSS $1 $2))
(LEQ $1 $2) :: (NOT (GTR $1 $2))
(EQL $1 $2) :: (NOT (NEQ $1 $2))
(NEQ $1 $2) :: (NOT (EQL $1 $2))
(GEQ $1 $2) :: (NOT (LSS $1 $2))
(GEQ $1 $2) :: (OR (EQL $1 $2) (GTR $1 $2))
(GTR $1 $2) :: (NOT (LEQ $1 $2))
(GTR $1 $2) :: (AND (NEQ $1 $2) (GEQ $1 $2))

!two's complement machines only!
(- $1) :: (+ (NOT $1) 1)
(* $1 2) :: (↑ $1 1)
(WHILE $1 $2) :: (;
                  (LABEL $3)
                  (-> (NOT $1) $4)
                  $2
                  (GOTO $3)
                  (LABEL $4) )

(IF $1 $2) :: (;   (-> (NOT $1) $3)
                  $2
                  (LABEL $3) )
(IF $1 $2 $3) :: (;
                  (-> $1 $4)
                  $3   (GOTO $5)
                  (LABEL $4)  $2
                  (LABEL $5) )

(GOTO $1) :: (← 7PC $1)
(IF $1 (← 7PC $2)) :: (-> $1 $2)

!the following are Simplifications for logical negation!
(NOT (NOT $1)) :: $1
(NOT (GTR $1 $2)) :: (LEQ $1 $2)        (NOT (LEQ $1 $2)) :: (GTR $1 $2)
(NOT (LSS $1 $2)) :: (GEQ $1 $2)        (NOT (GEQ $1 $2)) :: (LSS $1 $2)
(NOT (EQL $1 $2)) :: (NEQ $1 $2)        (NOT (NEQ $1 $2)) :: (EQL $1 $2)
```